目錄

Introduction 1.1
开发环境搭建 1.2
Node.js 的安装与使用 1.2.1
安装 Node.js 1.2.1.1
n 和 nvm 1.2.1.2
nrm 1.2.1.3
MongoDB 的安装与使用 1.2.2
安装与启动 MongoDB 1.2.2.1
Robomongo 和 MongoChef 1.2.2.2
Node.js 知识点讲解 1.3
require 1.3.1
exports 和 module.exports 1.3.2
Promise 1.3.3
环境变量 1.3.4
packge.json 1.3.5
semver 1.3.5.1
npm 使用注意事项 1.3.6
npm init 1.3.6.1
npm install 1.3.6.2
npm scripts 1.3.6.3
npm shrinkwrap 1.3.6.4
Hello, Express 1.4
初始化一个 Express 项目 1.4.1
supervisor 1.4.1.1
路由 1.4.2
express.Router 1.4.2.1
模板引擎 1.4.3

ejs 1.4.3.1
includes 1.4.3.2
Express 浅析 1.4.4
中间件与 next 1.4.4.1
错误处理 1.4.4.2
一个简单的博客 1.5
开发环境 1.5.1
准备工作 1.5.2
目录结构 1.5.2.1
安装依赖模块 1.5.2.2
配置文件 1.5.3
config-lite 1.5.3.1
功能设计 1.5.4
功能与路由设计 1.5.4.1
会话 1.5.4.2
页面通知 1.5.4.3
权限控制 1.5.4.4
页面设计 1.5.5
组件 1.5.5.1
app.locals 和 res.locals 1.5.5.2
连接数据库 1.5.6
为什么使用 Mongolass 1.5.6.1
注册 1.5.7
用户模型设计 1.5.7.1
注册页 1.5.7.2
注册与文件上传 1.5.7.3
登出与登录 1.5.8
登出 1.5.8.1
登录页 1.5.8.2
登录 1.5.8.3

文章 1.5.9
- 文章模型设计 1.5.9.1
- 发表文章 1.5.9.2
- 主页与文章页 1.5.9.3
- 编辑与删除文章 1.5.9.4

留言 1.5.10
- 留言模型设计 1.5.10.1
- 显示留言 1.5.10.2
- 发表与删除留言 1.5.10.3

404页面 1.5.11

错误页面 1.5.12

日志 1.5.13
- winston 和 express-winston 1.5.13.1
- .gitignore 1.5.13.2

测试 1.5.14
- mocha 和 supertest 1.5.14.1
- 测试覆盖率 1.5.14.2

部署 1.5.15
- 申请 MLab 1.5.15.1
- pm2 1.5.15.2
- 部署到 Heroku 1.5.15.3
- 部署到 UCloud 1.5.15.4

一起学 Node.js

> 作者：nswbmw
>
> 来源：N-blog

使用 Express + MongoDB 搭建多人博客

目录

- 开发环境搭建
  - Node.js 的安装与使用
    - 安装 Node.js
    - n 和 nvm
    - nrm
  - MongoDB 的安装与使用
    - 安装与启动 MongoDB
    - Robomongo 和 MongoChef
- Node.js 知识点讲解
  - require
  - exports 和 module.exports
  - Promise
  - 环境变量
  - packge.json
    - semver
  - npm 使用注意事项
    - npm init
    - npm install
    - npm scripts
    - npm shrinkwrap
- Hello, Express
  - 初始化一个 Express 项目
    - supervisor
  - 路由
    - express.Router

    - 模板引擎
        - ejs
        - includes
    - Express 浅析
        - 中间件与 next
        - 错误处理
- 一个简单的博客
    - 开发环境
    - 准备工作
        - 目录结构
        - 安装依赖模块
    - 配置文件
        - config-lite
    - 功能设计
        - 功能与路由设计
        - 会话
        - 页面通知
        - 权限控制
    - 页面设计
        - 组件
        - app.locals 和 res.locals
    - 连接数据库
        - 为什么使用 Mongolass
    - 注册
        - 用户模型设计
        - 注册页
        - 注册与文件上传
    - 登出与登录
        - 登出
        - 登录页
        - 登录
    - 文章
        - 文章模型设计
        - 发表文章
        - 主页与文章页
        - 编辑与删除文章

- 留言
  - 留言模型设计
  - 显示留言
  - 发表与删除留言
- 404页面
- 错误页面
- 日志
  - winston 和 express-winston
  - .gitignore
- 测试
  - mocha 和 supertest
  - 测试覆盖率
- 部署
  - 申请 MLab
  - pm2
  - 部署到 Heroku
  - 部署到 UCloud

捐赠

您的捐赠，是我持续开源的动力。

支付宝	微信

开发环境搭建

1.1.1 安装 Node.js

有三种方式安装 Node.js：一是通过安装包安装，二是通过源码编译安装，三是在 Linux 下可以通过 yum|apt-get 安装，在 Mac 下可以通过 Homebrew 安装。对于 Windows 和 Mac 用户，推荐使用安装包安装，Linux 用户推荐使用源码编译安装。

Windows 和 Mac 安装：

第一步：

打开 Node.js 官网，可以看到以下两个下载选项：



左边的是 LTS 版，用过 ubuntu 的同学可能比较熟悉，即长期支持版本，大多数人用这个就可以了。右边是最新版，支持最新的语言特性（比如对 ES6 的支持更全面），想尝试新特性的开发者可以安装这个版本。我们选择左边的 v6.9.1 LTS 点击下载。

> 小提示：从 http://node.green 上可以看到 Node.js 各个版本对 ES6 的支持情况。

第二步：

安装 Node.js，这个没什么好说的，一直点击 `继续` 即可。



第三步：

提示安装成功后，打开终端输入以下命令，可以看到 node 和 npm 都已经安装好了：

```
➜ myblog node -v
v6.9.1
➜ myblog npm -v
3.10.8
```

Linux 安装：

Linux 用户可通过源码编译安装：

```
curl -O https://nodejs.org/dist/v6.9.1/node-v6.9.1.tar.gz
tar -xzvf node-v6.9.1.tar.gz
cd node-v6.9.1
./configure
make
make install
```

> 注意: 如果编译过程报错，可能是缺少某些依赖包。因为报错内容不尽相同，请读者自行求助搜索引擎或 stackoverflow。

1.1.2 n 和 nvm

通常我们使用稳定的 LTS 版本的 Node.js 即可，但有的情况下我们又想尝试一下新的特性，我们总不能来回安装不同版本的 Node.js 吧，这个时候我们就需要 n 或者 nvm 了。n 和 nvm 是两个常用的 Node.js 版本管理工具，关于 n 和 nvm 的使用以及区别，这篇文章 讲得特别详细，这里不再赘述。

1.1.3 nrm

nrm 是一个管理 npm 源的工具。用过 ruby 和 gem 的同学会比较熟悉，通常我们会把 gem 源切到国内的淘宝镜像，这样在安装和更新一些包的时候比较快。nrm 同理，用来切换官方 npm 源和国内的 npm 源（如: cnpm），当然也可以用来切换官方 npm 源和公司私有 npm 源。

全局安装 nrm:

```
npm i nrm -g
```

查看当前 nrm 内置的几个 npm 源的地址：

```
➜ Desktop nrm ls

* npm ---- https://registry.npmjs.org/
  cnpm --- http://r.cnpmjs.org/
  taobao - https://registry.npm.taobao.org/
  nj ----- https://registry.nodejitsu.com/
  rednpm - http://registry.mirror.cqupt.edu.cn/
  npmMirror  https://skimdb.npmjs.com/registry/
  edunpm - http://registry.enpmjs.org/
```

切换到 cnpm：

```
➜ Desktop nrm use cnpm


   Registry has been set to: http://r.cnpmjs.org/

➜ Desktop nrm ls

  npm ---- https://registry.npmjs.org/
* cnpm --- http://r.cnpmjs.org/
  taobao - https://registry.npm.taobao.org/
  nj ----- https://registry.nodejitsu.com/
  rednpm - http://registry.mirror.cqupt.edu.cn/
  npmMirror  https://skimdb.npmjs.com/registry/
  edunpm - http://registry.enpmjs.org/
```

下一节：1.2 MongoDB 的安装与使用

1.2.1 安装与启动 MongoDB

- Windows 用户向导：https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/
- Linux 用户向导：https://docs.mongodb.com/manual/administration/install-on-linux/
- Mac 用户向导：https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/

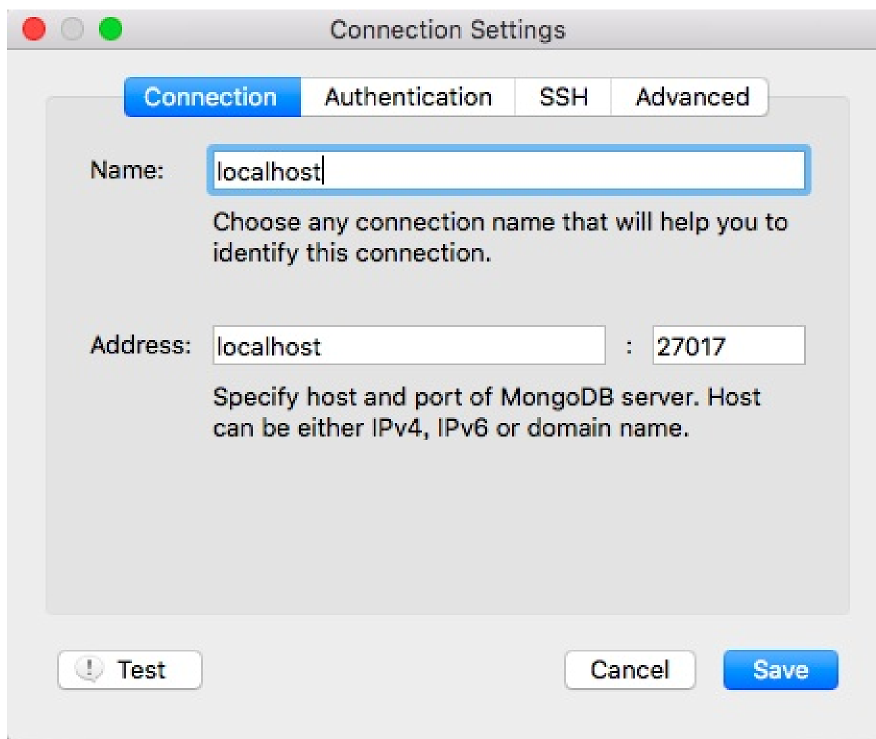
1.2.2 Robomongo 和 Mongochef

Robomongo

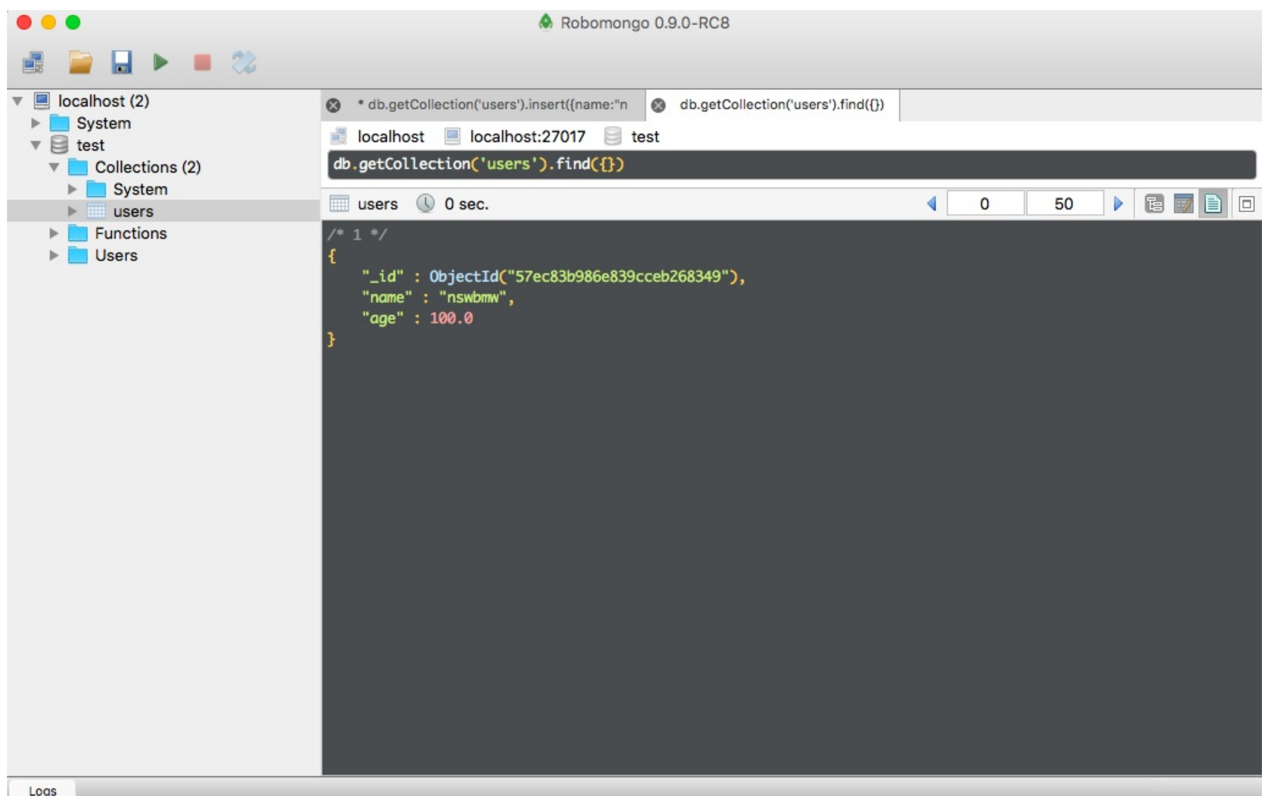
Robomongo 是一个基于 Shell 的跨平台开源 MongoDB 可视化管理工具，支持 Windows、Linux 和 Mac，嵌入了 JavaScript 引擎和 MongoDB mongo，只要你会使用 mongo shell，你就会使用 Robomongo，它还提了供语法高亮、自动补全、差别视图等。

Robomongo 下载地址

下载并安装成功后点击左上角的 `Create` 创建一个连接，给该连接起个名字如: `localhost` ，使用默认地址（localhost）和端口（27017）即可，点击 `Save` 保存。



双击 `localhost` 连接到 MongoDB 并进入交互界面，尝试插入一条数据并查询出来，如下所示:



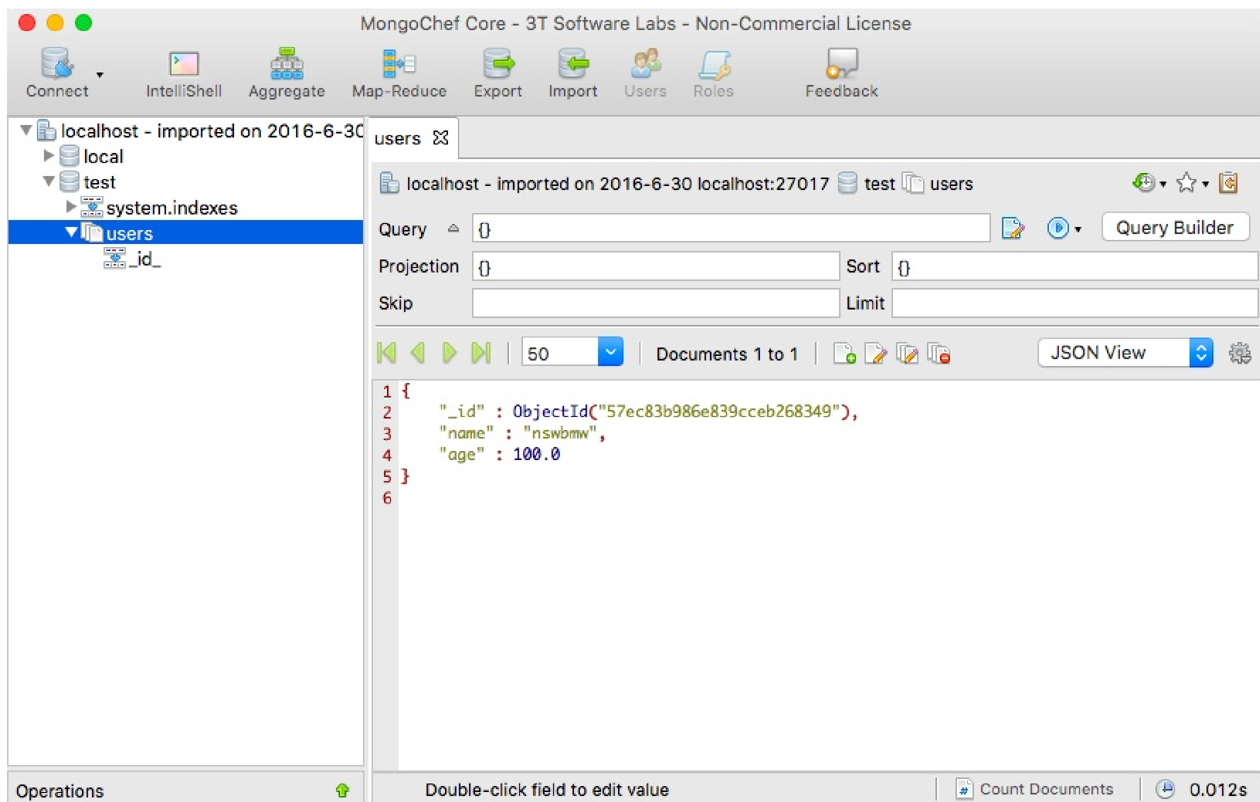
MongoChef

MongoChef 是另一款强大的 MongoDB 可视化管理工具，支持 Windows、Linux 和 Mac。

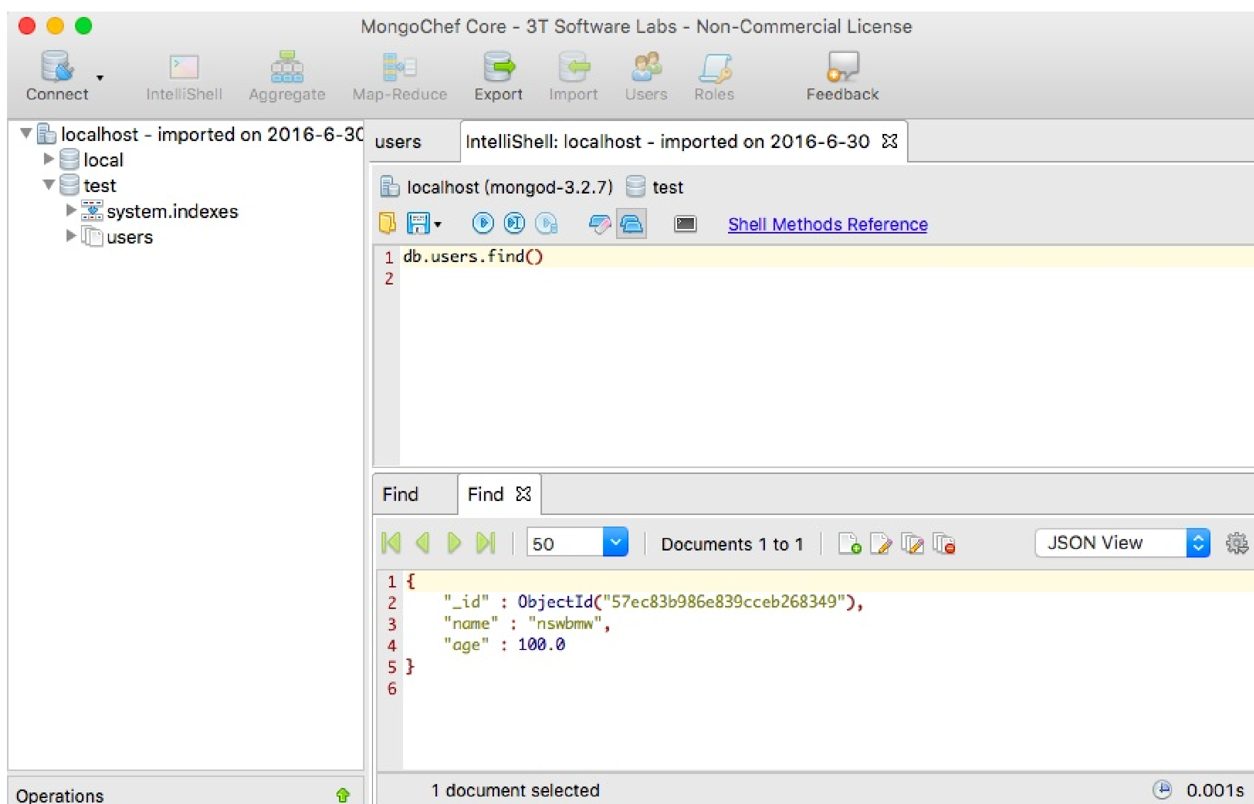
MongoChef 下载地址，我们选择左侧的非商业用途的免费版下载。

	MongoChef Free For Non-Commercial Use	MongoChef For Commercial Use
Operating System	Windows, Mac, Linux	Windows, Mac, Linux
MongoDB Versions	2.4, 2.6, 3.0, 3.2	2.4, 2.6, 3.0, 3.2
Fully Featured MongoDB GUI (See Details)	✔	✔
Optional Priority Support	–	✔
Support for MongoDB Enterprise	–	✔
Kerberos (GSSAPI) authentication	–	✔
LDAP authentication	–	✔
	Download MongoChef Core Free download	Download MongoChef Free 14-day trial

安装成功后跟 Robomongo 一样，也需要创建一个新的连接的配置，成功后双击进入到 MongoChef 主页面，如下所示:



还可以使用 shell 模式:



> 小提示: MongoChef 相较于 Robomongo 更强大一些，但 Robomongo 比较轻量也能满足大部分的常规需求，所以哪一个适合自己还需读者自行尝试。

上一节：1.1 Node.js 的安装与使用

下一节：2.1 require

Node.js 知识点讲解

require 用来加载一个文件的代码，关于 require 的机制这里不展开讲解，请仔细阅读 官方文档 。

简单概括以下几点:

- require 可加载 .js、.json 和 .node 后缀的文件
- require 的过程是同步的，所以这样是错误的:

  ```
  setTimeout(() => {
  module.exports = { a: 'hello' };
  }, 0);
  ```

  require 这个文件得到的是空对象 `{}`

- require 目录的机制是:
  - 如果目录下有 package.json 并指定了 main 字段，则用之
  - 如果不存在 package.json，则依次尝试加载目录下的 index.js 和 index.node
- require 过的文件会加载到缓存，所以多次 require 同一个文件（模块）不会重复加载
- 判断是否是程序的入口文件有两种方式:
  - require.main === module（推荐）
  - module.parent === null

循环引用

循环引用（或循环依赖）简单点来说就是 a 文件 require 了 b 文件，然后 b 文件又反过来 require 了 a 文件。我们用 a->b 代表 b require 了 a。

简单的情况:

```
a->b
b->a
```

复杂点的情况:

```
a->b
b->c
c->a
```

循环引用并不会报错，导致的结果是 require 的结果是空对象 `{}` ，原因是 b require 了 a，a 又去 require 了 b，此时 b 还没初始化好，所以只能拿到初始值 `{}` 。当产生循环引用时一般有两种方法解决：

1. 通过分离共用的代码到另一个文件解决，如上面简单的情况，可拆出共用的代码到 c 中，如下:

```
c->a
c->b
```

1. 不在最外层 require，在用到的地方 require，通常在函数的内部

总的来说，循环依赖的陷阱并不大容易出现，但一旦出现了，对于新手来说还真不好定位。它的存在给我们提了个醒，要时刻注意你项目的依赖关系不要过于复杂，哪天你发现一个你明明已经 exports 了的方法报 `undefined is not a function` ，我们就该提醒一下自己：哦，也许是它来了。

官方示例: https://nodejs.org/api/modules.html#modules_cycles

上一节：1.2 MongoDB 的安装与使用

下一节：2.2 exports 和 module.exports

require 用来加载代码，而 exports 和 module.exports 则用来导出代码。

很多新手可能会迷惑于 exports 和 module.exports 的区别，为了更好的理解 exports 和 module.exports 的关系，我们先来巩固下 js 的基础。示例：

test.js

```
var a = {name: 1};
var b = a;

console.log(a);
console.log(b);

b.name = 2;
console.log(a);
console.log(b);

var b = {name: 3};
console.log(a);
console.log(b);
```

运行 test.js 结果为：

```
{ name: 1 }
{ name: 1 }
{ name: 2 }
{ name: 2 }
{ name: 2 }
{ name: 3 }
```

解释：a 是一个对象，b 是对 a 的引用，即 a 和 b 指向同一块内存，所以前两个输出一样。当对 b 作修改时，即 a 和 b 指向同一块内存地址的内容发生了改变，所以 a 也会体现出来，所以第三四个输出一样。当 b 被覆盖时，b 指向了一块新的内存，a 还是指向原来的内存，所以最后两个输出不一样。

明白了上述例子后，我们只需知道三点就知道 exports 和 module.exports 的区别了：

1. module.exports 初始值为一个空对象 {}

2. exports 是指向的 module.exports 的引用
3. require() 返回的是 module.exports 而不是 exports

Node.js 官方文档的截图证实了我们的观点:

exports alias #

Added in: v0.1.16

The `exports` variable that is available within a module starts as a reference to `module.exports`. As with any variable, if you assign a new value to it, it is no longer bound to the previous value.

To illustrate the behavior, imagine this hypothetical implementation of `require()`:

```
function require(...) {
  // ...
  ((module, exports) => {
    // Your module code here
    exports = some_func;         // re-assigns exports, exports is no longer
                                 // a shortcut, and nothing is exported.
    module.exports = some_func; // makes your module export 0
  })(module, module.exports);
  return module;
}
```

As a guideline, if the relationship between `exports` and `module.exports` seems like magic to you, ignore `exports` and only use `module.exports`.

exports = module.exports = {...}

我们经常看到这样的写法：

```
exports = module.exports = {...}
```

上面的代码等价于:

```
module.exports = {...}
exports = module.exports
```

原理很简单：module.exports 指向新的对象时，exports 断开了与 module.exports 的引用，那么通过 exports = module.exports 让 exports 重新指向 module.exports。

> 小提示：ES6 的 import 和 export 不在本文的讲解范围，有兴趣的读者可以去学习阮一峰老师的《ECMAScript6入门》。

上一节：2.1 require

下一节：2.3 Promise

网上已经有许多关于 Promise 的资料了，这里不在赘述。以下 4 个链接供读者学习：

1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise （基础）
2. http://liubin.org/promises-book/ （开源 Promise 迷你书）
3. http://fex.baidu.com/blog/2015/07/we-have-a-problem-with-promises/ （进阶）
4. https://promisesaplus.com/ （官方定义规范）

Promise 用于异步流程控制，生成器与 yield 也能实现流程控制（基于 co），但不在本教程讲解范围内，读者可参考我的另一部教程 N-club。async/await 结合 Promise 也可以实现流程控制，有兴趣请查阅 《ECMAScript6入门》。

上一节：2.2 exports 和 module.exports

下一节：2.4 环境变量

环境变量不属于 Node.js 的知识范畴，只不过我们在开发 Node.js 应用时经常与环境变量打交道，所以这里简单介绍下。

环境变量（environment variables）一般是指在操作系统中用来指定操作系统运行环境的一些参数。在 Mac 和 Linux 的终端直接输入 env，会列出当前的环境变量，如：USER=xxx。简单来讲，环境变量就是传递参数给运行程序的。

在 Node.js 中，我们经常这么用:

```
NODE_ENV=test node app
```

通过以上命令启动程序，指定当前环境变量 `NODE_ENV` 的值为 test，那么在 app.js 中可通过 `process.env` 来获取环境变量:

```
console.log(process.env.NODE_ENV) //test
```

另一个常见的例子是使用 debug 模块时:

```
DEBUG=* node app
```

Windows 用户需要首先设置环境变量，然后再执行程序：

```
set DEBUG=*
set NODE_ENV=test
node app
```

或者使用 cross-env：

```
npm i cross-env -g
```

使用方式：

```
cross-env NODE_ENV=test node app
```

上一节：2.3 Promise

下一节：2.5 packge.json

package.json 对于 Node.js 应用来说是一个不可或缺的文件，它存储了该 Node.js 应用的名字、版本、描述、作者、入口文件、脚本、版权等等信息。npm 官网有 package.json 每个字段的详细介绍：https://docs.npmjs.com/files/package.json。

2.5.1 semver

语义化版本（semver）即 dependencies、devDependencies 和 peerDependencies 里的如： `"co": "^4.6.0"` 。

semver 格式： `主版本号.次版本号.修订号` 。版本号递增规则如下：

- `主版本号` ：做了不兼容的 API 修改
- `次版本号` ：做了向下兼容的功能性新增
- `修订号` ：做了向下兼容的 bug 修正

更多阅读：

1. http://semver.org/lang/zh-CN/
2. http://taobaofed.org/blog/2016/08/04/instructions-of-semver/

作为 Node.js 的开发者，我们在发布 npm 模块的时候一定要遵守语义化版本的命名规则，即：有 breaking change 发大版本，有新增的功能发小版本，有小的 bug 修复或优化则发修订版本。

上一节：2.4 环境变量

下一节：2.6 npm 使用注意事项

2.6.1 npm init

使用 `npm init` 初始化一个空项目是一个好的习惯，即使你对 package.json 及其他属性非常熟悉， `npm init` 也是你开始写新的 Node.js 应用或模块的一个快捷的办法。 `npm init` 有智能的默认选项，比如从根目录名称推断模块名称，通过 `~/.npmrc` 读取你的信息，用你的 Git 设置来确定 repository 等等。

2.6.2 npm install

`npm install` 是我们最常用的 npm 命令之一，因此我们需要好好了解下这个命令。终端输入 `npm install -h` 查看使用方式:

```
➜ Desktop npm install -h
npm install
npm install <pkg>
npm install <pkg>@<tag>
npm install <pkg>@<version>
npm install <pkg>@<version range>
npm install <folder>
npm install <tarball file>
npm install <tarball url>
npm install <git:// url>
npm install <github username>/<github project>

Can specify one or more: npm install ./foo.tgz bar@stable /some/folder
If no argument is supplied and ./npm-shrinkwrap.json is
present, installs dependencies specified in the shrinkwrap.
Otherwise, installs dependencies from ./package.json.
```

可以看出：我们通过 `npm install` 可以安装 npm 上发布的某个版本、某个tag、某个版本区间的模块，甚至可以安装本地目录、压缩包和 git/github 的库作为依赖。

> 小提示: `npm i` 是 `npm install` 的简写，建议使用 `npm i` 。

直接使用 `npm i` 安装的模块是不会写入 package.json 的 dependencies (或 devDependencies)，需要额外加个参数:

1. `npm i express --save` / `npm i express -S` (安装 express，同时将 `"express": "^4.14.0"` 写入 dependencies)
2. `npm i express --save-dev` / `npm i express -D` (安装 express，同时将 `"express": "^4.14.0"` 写入 devDependencies)

3. `npm i express --save --save-exact` (安装 express，同时将 `"express": "4.14.0"` 写入 dependencies)

第三种方式将固定版本号写入 dependencies，建议线上的 Node.js 应用都采取这种锁定版本号的方式，因为你不可能保证第三方模块下个小版本是没有验证 bug 的，即使是很流行的模块。拿 Mongoose 来说，Mongoose 4.1.4 引入了一个 bug 导致调用一个文档 entry 的 remove 会删除整个集合的文档，见：https://github.com/Automattic/mongoose/blob/master/History.md#415--2015-09-01 。

> 后面会介绍更安全的 `npm shrinkwrap` 的用法。

运行以下命令：

```
npm config set save-exact true
```

这样每次 `npm i xxx --save` 的时候会锁定依赖的版本号，相当于加了 `--save-exact` 参数。

> 小提示：`npm config set` 命令将配置写到了 ~/.npmrc 文件，运行 `npm config list` 查看。

2.6.3 npm scripts

npm 提供了灵活而强大的 scripts 功能，见 官方文档 。

npm 的 scripts 有一些内置的缩写命令，如常用的：

- `npm start` 等价于 `npm run start`
- `npm test` 等价于 `npm run test`

2.6.4 npm shrinkwrap

前面说过要锁定依赖的版本，但这并不能完全防止意外情况的发生，因为锁定的只是最外一层的依赖，而里层依赖的模块的 package.json 有可能写的是 `"mongoose": "*"` 。为了彻底锁定依赖的版本，让你的应用在任何机器上安装的都是同样版本的模块（不管嵌套多少层），通过运行 `npm shrinkwrap` ，会在当前目录下产生一个 `npm-shrinkwrap.json` ，里面包含了通过 node_modules 计

算出的模块的依赖树及版本。上面的截图也显示：只要目录下有 npm-shrinkwrap.json 则运行 `npm install` 的时候会优先使用 npm-shrinkwrap.json 进行安装，没有则使用 package.json 进行安装。

更多阅读：

1. https://docs.npmjs.com/cli/shrinkwrap
2. http://tech.meituan.com/npm-shrinkwrap.html

> 注意: 如果 node_modules 下存在某个模块（如直接通过 `npm install xxx` 安装的）而 package.json 中没有，运行 `npm shrinkwrap` 则会报错。另外，`npm shrinkwrap` 只会生成 dependencies 的依赖，不会生成 devDependencies 的。

上一节：2.5 packge.json

下一节：3.1 初始化一个 Express 项目

Hello, Express!

首先，我们新建一个目录 myblog，在该目录下运行 `npm init` 生成一个 package.json，如下所示：

```
➜ Desktop mkdir myblog
➜ Desktop cd myblog
➜ myblog npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (myblog)
version: (1.0.0)
description: my first blog
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/nswbmw/Desktop/myblog/package.json:

{
  "name": "myblog",
  "version": "1.0.0",
  "description": "my first blog",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this ok? (yes) y
```

> 注意：括号里的是默认值，如果使用默认值则直接回车即可，否则输入自定义内容后回车。

然后安装 express 并写入 package.json：

```
npm i express@4.14.0 --save
```

新建 index.js，添加如下代码：

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
  res.send('hello, express');
});

app.listen(3000);
```

以上代码的意思是：生成一个 express 实例 app，挂载了一个根路由控制器，然后监听 3000 端口并启动程序。运行 `node index`，打开浏览器访问 `localhost:3000` 时，页面应显示 hello, express。

这是最简单的一个使用 express 的例子，后面会介绍路由及模板的使用。

3.1.1 supervisor

在开发过程中，每次修改代码保存后，我们都需要手动重启程序，才能查看改动的效果。使用 supervisor 可以解决这个繁琐的问题，全局安装 supervisor：

```
npm install -g supervisor
```

运行 `supervisor --harmony index` 启动程序，如下所示：

```
➜ myblog supervisor --harmony index

Running node-supervisor with
  program '--harmony index'
  --watch '.'
  --extensions 'node,js'
  --exec 'node'

Starting child process with 'node --harmony index'
Watching directory '/Users/nswbmw/Desktop/myblog' for changes.
Press rs for restarting the process.
```

supervisor 会监听当前目录下 node 和 js 后缀的文件，当这些文件发生改动时，supervisor 会自动重启程序。

上一节：2.6 npm 使用注意事项

下一节：3.2 路由

前面我们只是挂载了根路径的路由控制器，现在修改 index.js 如下：

```
var express = require('express');
var app = express();

app.get('/', function(req, res) {
  res.send('hello, express');
});

app.get('/users/:name', function(req, res) {
  res.send('hello, ' + req.params.name);
});

app.listen(3000);
```

以上代码的意思是：当访问根路径时，依然返回 hello, express，当访问如 `localhost:3000/users/nswbmw` 路径时，返回 hello, nswbmw。路径中 `:name` 起了占位符的作用，这个占位符的名字是 name，可以通过 `req.params.name` 取到实际的值。

> 小提示：express 使用了 path-to-regexp 模块实现的路由匹配。

不难看出：req 包含了请求来的相关信息，res 则用来返回该请求的响应，更多请查阅 express 官方文档。下面介绍几个常用的 req 的属性：

- `req.query`：解析后的 url 中的 querystring，如 `?name=haha`，req.query 的值为 `{name: 'haha'}`
- `req.params`：解析 url 中的占位符，如 `/:name`，访问 /haha，req.params 的值为 `{name: 'haha'}`
- `req.body`：解析后请求体，需使用相关的模块，如 body-parser，请求体为 `{"name": "haha"}`，则 req.body 为 `{name: 'haha'}`

3.2.1 express.Router

上面只是很简单的路由使用的例子（将所有路由控制函数都放到了 index.js），但在实际开发中通常有几十甚至上百的路由，都写在 index.js 既臃肿又不好维护，这时可以使用 express.Router 实现更优雅的路由解决方案。在 myblog 目录下创建空文件夹 routes，在 routes 目录下创建 index.js 和 users.js。最后代码如下：

index.js

```
var express = require('express');
var app = express();
var indexRouter = require('./routes/index');
var userRouter = require('./routes/users');

app.use('/', indexRouter);
app.use('/users', userRouter);

app.listen(3000);
```

routes/index.js

```
var express = require('express');
var router = express.Router();

router.get('/', function(req, res) {
  res.send('hello, express');
});

module.exports = router;
```

routes/users.js

```
var express = require('express');
var router = express.Router();

router.get('/:name', function(req, res) {
  res.send('hello, ' + req.params.name);
});

module.exports = router;
```

以上代码的意思是：我们将 `/` 和 `/users/:name` 的路由分别放到了 routes/index.js 和 routes/users.js 中，每个路由文件通过生成一个 express.Router 实例 router 并导出，通过 `app.use` 挂载到不同的路径。这两种代码实现了相同

的功能，但在实际开发中推荐使用 express.Router 将不同的路由分离到不同的路由文件中。

更多 express.Router 的用法见 express 官方文档。

上一节：3.1 初始化一个 Express 项目

下一节：3.3 模板引擎

模板引擎（Template Engine）是一个将页面模板和数据结合起来生成 html 的工具。上例中，我们只是返回纯文本给浏览器，现在我们修改代码返回一个 html 页面给浏览器。

3.3.1 ejs

模板引擎有很多，ejs 是其中一种，因为它使用起来十分简单，而且与 express 集成良好，所以我们使用 ejs。安装 ejs：

```
npm i ejs --save
```

修改 index.js 如下：

index.js

```
var path = require('path');
var express = require('express');
var app = express();
var indexRouter = require('./routes/index');
var userRouter = require('./routes/users');

app.set('views', path.join(__dirname, 'views'));// 设置存放模板文件的目录
app.set('view engine', 'ejs');// 设置模板引擎为 ejs

app.use('/', indexRouter);
app.use('/users', userRouter);

app.listen(3000);
```

通过 `app.set` 设置模板引擎为 ejs 和存放模板的目录。在 myblog 下新建 views 文件夹，在 views 下新建 users.ejs，添加如下代码：

views/users.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <style type="text/css">
      body {padding: 50px;font: 14px "Lucida Grande", Helvetica,
 Arial, sans-serif;}
    </style>
  </head>
  <body>
    <h1><%= name.toUpperCase() %></h1>
    <p>hello, <%= name %></p>
  </body>
</html>
```

修改 routes/users.js 如下：

routes/users.js

```
var express = require('express');
var router = express.Router();

router.get('/:name', function(req, res) {
  res.render('users', {
    name: req.params.name
  });
});

module.exports = router;
```

通过调用 `res.render` 函数渲染 ejs 模板，res.render 第一个参数是模板的名字，这里是 users 则会匹配 views/users.ejs，第二个参数是传给模板的数据，这里传入 name，则在 ejs 模板中可使用 name。 `res.render` 的作用就是将模板和数据结合生成 html，同时设置响应头中的 `Content-Type: text/html` ，告诉浏览器我返回的是 html，不是纯文本，要按 html 展示。现在我们访问 `localhost:3000/users/haha` ，如下图所示：

HAHA

hello, haha

上面代码可以看到，我们在模板 `<%= name.toUpperCase() %>` 中使用了 JavaScript 的语法 `.toUpperCase()` 将名字转化为大写，那这个 `<%= xxx %>` 是什么东西呢？ejs 有 3 种常用标签：

1. `<% code %>` ：运行 JavaScript 代码，不输出
2. `<%= code %>` ：显示转义后的 HTML内容
3. `<%- code %>` ：显示原始 HTML 内容

> 注意： `<%= code %>` 和 `<%- code %>` 都可以是 JavaScript 表达式生成的字符串，当变量 code 为普通字符串时，两者没有区别。当 code 比如为 `<h1>hello</h1>` 这种字符串时， `<%= code %>` 会原样输出 `<h1>hello</h1>` ，而 `<%- code %>` 则会显示 H1 大的 hello 字符串。

下面的例子解释了 `<% code %>` 的用法：

Data

```
supplies: ['mop', 'broom', 'duster']
```

Template

```
<ul>
<% for(var i=0; i<supplies.length; i++) {%>
   <li><%= supplies[i] %></li>
<% } %>
</ul>
```

Result

```
<ul>
  <li>mop</li>
  <li>broom</li>
  <li>duster</li>
</ul>
```

更多 ejs 的标签请看 官方文档。

3.3.2 includes

我们使用模板引擎通常不是一个页面对应一个模板，这样就失去了模板的优势，而是把模板拆成可复用的模板片段组合使用，如在 views 下新建 header.ejs 和 footer.ejs，并修改 users.ejs：

views/header.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <style type="text/css">
      body {padding: 50px;font: 14px "Lucida Grande", Helvetica,
 Arial, sans-serif;}
    </style>
  </head>
  <body>
```

views/footer.ejs

```
  </body>
</html>
```

views/users.ejs

```
<%- include('header') %>
  <h1><%= name.toUpperCase() %></h1>
  <p>hello, <%= name %></p>
<%- include('footer') %>
```

我们将原来的 users.ejs 拆成出了 header.ejs 和 footer.ejs，并在 users.ejs 通过 ejs 内置的 include 方法引入，从而实现了跟以前一个模板文件相同的功能。

> 小提示：拆分模板组件通常有两个好处：
>
> 1. 模板可复用，减少重复代码
> 2. 主模板结构清晰
>
> 注意：要用 `<%- include('header') %>` 而不是 `<%= include('header') %>`

上一节：3.2 路由

下一节：3.4 Express 浅析

前面我们讲解了 express 中路由和模板引擎 ejs 的用法，但 express 的精髓并不在此，在于中间件的设计理念。

3.4.1 中间件与 next

express 中的中间件（middleware）就是用来处理请求的，当一个中间件处理完，可以通过调用 `next()` 传递给下一个中间件，如果没有调用 `next()` ，则请求不会往下传递，如内置的 `res.render` 其实就是渲染完 html 直接返回给客户端，没有调用 `next()` ，从而没有传递给下一个中间件。看个小例子，修改 index.js 如下：

index.js

```
var express = require('express');
var app = express();

app.use(function(req, res, next) {
  console.log('1');
  next();
});

app.use(function(req, res, next) {
  console.log('2');
  res.status(200).end();
});

app.listen(3000);
```

此时访问 `localhost:3000` ，终端会输出：

```
1
2
```

通过 `app.use` 加载中间件，在中间件中通过 next 将请求传递到下一个中间件，next 可接受一个参数接收错误信息，如果使用了 `next(error)` ，则会返回错误而不会传递到下一个中间件，修改 index.js 如下：

index.js

```
var express = require('express');
var app = express();

app.use(function(req, res, next) {
  console.log('1');
  next(new Error('haha'));
});

app.use(function(req, res, next) {
  console.log('2');
  res.status(200).end();
});

app.listen(3000);
```

此时访问 `localhost:3000` ，终端会输出错误信息：

```
1
Error: haha
    at /Users/nswbmw/Desktop/myblog/index.js:6:8
    at Layer.handle [as handle_request] (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/layer.js:95:5)
    at trim_prefix (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:312:13)
    at /Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:280:7
    at Function.process_params (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:330:12)
    at next (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:271:10)
    at expressInit (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/middleware/init.js:33:5)
    at Layer.handle [as handle_request] (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/layer.js:95:5)
    at trim_prefix (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:312:13)
    at /Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:280:7
```

浏览器会显示：

```
Error: haha
    at /Users/nswbmw/Desktop/myblog/index.js:6:8
    at Layer.handle [as handle_request] (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/layer.js:95:5)
    at trim_prefix (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:312:13)
    at /Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:280:7
    at Function.process_params (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:330:12)
    at next (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:271:10)
    at expressInit (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/middleware/init.js:33:5)
    at Layer.handle [as handle_request] (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/layer.js:95:5)
    at trim_prefix (/Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:312:13)
    at /Users/nswbmw/Desktop/myblog/node_modules/express/lib/router/index.js:280:7
```

> 小提示： `app.use` 有非常灵活的使用方式，详情见 官方文档。

express 有成百上千的第三方中间件，在开发过程中我们首先应该去 npm 上寻找是否有类似实现的中间件，尽量避免造轮子，节省开发时间。下面给出几个常用的搜索 npm 模块的网站：

1. http://npmjs.com(npm 官网)
2. http://node-modules.com
3. https://npms.io
4. https://nodejsmodules.org

> 小提示：express@4 之前的版本基于 connect 这个模块实现的中间件的架构，express@4 及以上的版本则移除了对 connect 的依赖自己实现了，理论上基于 connect 的中间件（通常以 `connect-` 开头，如 `connect-mongo`）仍可结合 express 使用。
>
> 注意：中间件的加载顺序很重要！比如：通常把日志中间件放到比较靠前的位置，后面将会介绍的 `connect-flash` 中间件是基于 session 的，所以需要在 `express-session` 后加载。

3.4.2 错误处理

上面的例子中，应用程序为我们自动返回了错误栈信息（express 内置了一个默认的错误处理器），假如我们想手动控制返回的错误内容，则需要加载一个自定义错误处理的中间件，修改 index.js 如下：

index.js

```
var express = require('express');
var app = express();

app.use(function(req, res, next) {
  console.log('1');
  next(new Error('haha'));
});

app.use(function(req, res, next) {
  console.log('2');
  res.status(200).end();
});

//错误处理
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});

app.listen(3000);
```

此时访问 `localhost:3000` ，浏览器会显示 `Something broke!` 。

> 小提示：关于 express 的错误处理，详情见 官方文档。

上一节：3.3 模板引擎

下一节：4.1 开发环境

一个简单的博客

从本章开始，正式学习如何使用 Express + MongoDB 搭建一个博客。

Node.js: `6.9.1`

MongoDB: `3.2.10`

Express: `4.14.0`

上一节：3.4 Express 浅析

下一节：4.2 准备工作

4.2.1 目录结构

我们停止 supervisor 并删除 myblog 目录从头来过。重新创建 myblog，运行 `npm init` ，如下：

```
➜ Desktop mkdir myblog
➜ Desktop cd myblog
➜ myblog npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

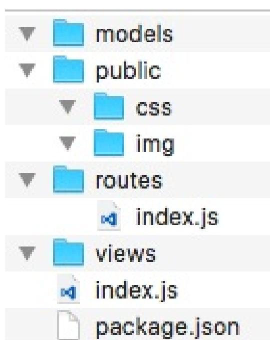
Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (myblog)
version: (1.0.0)
description: my first blog
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/nswbmw/Desktop/myblog/package.json:

{
  "name": "myblog",
  "version": "1.0.0",
  "description": "my first blog",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}


Is this ok? (yes) y
```

在 myblog 目录下创建以下目录及空文件（package.json 除外）：



对应文件及文件夹的用处：

1. `models`：存放操作数据库的文件
2. `public`：存放静态文件文件，如样式、图片等
3. `routes`：存放路由文件
4. `views`：存放模板文件
5. `index.js`：程序主文件
6. `package.json`：存储项目名、描述、作者、依赖等等信息

> 小提示：不知读者发现了没有，我们遵循了 MVC（模型(model)－视图(view)－控制器(controller/route)）的开发模式。

4.2.2 安装依赖模块

运行以下命令安装所需模块：

```
npm i config-lite connect-flash connect-mongo ejs express express-formidable express-session marked moment mongolass objectid-to-timestamp sha1 winston express-winston --save
```

对应模块的用处：

1. `express`：web 框架
2. `express-session`：session 中间件
3. `connect-mongo`：将 session 存储于 mongodb，结合 express-session 使用
4. `connect-flash`：页面通知提示的中间件，基于 session 实现
5. `ejs`：模板
6. `express-formidable`：接收表单及文件的上传中间件
7. `config-lite`：读取配置文件
8. `marked`：markdown 解析

9. `moment`：时间格式化
10. `mongolass`：mongodb 驱动
11. `objectid-to-timestamp`：根据 ObjectId 生成时间戳
12. `sha1`：sha1 加密，用于密码加密
13. `winston`：日志
14. `express-winston`：基于 winston 的用于 express 的日志中间件

后面会详细讲解这些模块的用处。

上一节：4.1 开发环境

下一节：4.3 配置文件

不管是小项目还是大项目，将配置与代码分离是一个非常好的做法。我们通常将配置写到一个配置文件里，如 config.js 或 config.json ，并放到项目的根目录下。但通常我们都会有许多环境，如本地开发环境、测试环境和线上环境等，不同的环境的配置不同，我们不可能每次部署时都要去修改引用 config.test.js 或者 config.production.js。config-lite 模块正是你需要的。

4.3.1 config-lite

config-lite 是一个轻量的读取配置文件的模块。config-lite 会根据环境变量（`NODE_ENV`）的不同从当前执行进程目录下的 config 目录加载不同的配置文件。如果不设置 `NODE_ENV`，则读取默认的 default 配置文件，如果设置了 `NODE_ENV`，则会合并指定的配置文件和 default 配置文件作为配置，config-lite 支持 .js、.json、.node、.yml、.yaml 后缀的文件。

如果程序以 `NODE_ENV=test node app` 启动，则通过 `require('config-lite')` 会依次降级查找 `config/test.js`、`config/test.json`、`config/test.node`、`config/test.yml`、`config/test.yaml` 并合并 default 配置; 如果程序以 `NODE_ENV=production node app` 启动，则通过 `require('config-lite')` 会依次降级查找 `config/production.js`、`config/production.json`、`config/production.node`、`config/production.yml`、`config/production.yaml` 并合并 default 配置。

在 myblog 下新建 config 目录，在该目录下新建 default.js，添加如下代码：

config/default.js

```
module.exports = {
  port: 3000,
  session: {
    secret: 'myblog',
    key: 'myblog',
    maxAge: 2592000000
  },
  mongodb: 'mongodb://localhost:27017/myblog'
};
```

配置释义：

1. `port`：程序启动要监听的端口号
2. `session`：express-session 的配置信息，后面介绍
3. `mongodb`：mongodb 的地址，`myblog` 为 db 名

上一节：4.2 准备工作

下一节：4.4 功能设计

4.4.1 功能与路由设计

在开发博客之前，我们首先需要明确博客要实现哪些功能。由于本教程面向初学者，所以只实现了博客最基本的功能，其余的功能（如归档、标签、分页等等）读者可自行实现。

功能及路由设计如下：

1. 注册
   i. 注册页：`GET /signup`
   ii. 注册（包含上传头像）：`POST /signup`
2. 登录
   i. 登录页：`GET /signin`
   ii. 登录：`POST /signin`
3. 登出：`GET /signout`
4. 查看文章
   i. 主页：`GET /posts`
   ii. 个人主页：`GET /posts?author=xxx`
   iii. 查看一篇文章（包含留言）：`GET /posts/:postId`
5. 发表文章
   i. 发表文章页：`GET /posts/create`
   ii. 发表文章：`POST /posts`
6. 修改文章
   i. 修改文章页：`GET /posts/:postId/edit`
   ii. 修改文章：`POST /posts/:postId/edit`
7. 删除文章：`GET /posts/:postId/remove`
8. 留言
   i. 创建留言：`POST /posts/:postId/comment`
   ii. 删除留言：`GET /posts/:postId/comment/:commentId/remove`

由于我们博客页面是后端渲染的，所以只通过简单的 `<a>(GET)` 和 `<form>(POST)` 与后端进行交互，如果使用 jQuery 或者其他前端框架（如 angular、vue、react 等等）可通过 Ajax 与后端交互，则 api 的设计应尽量遵循 restful 风格。

restful

restful 是一种 api 的设计风格，提出了一组 api 的设计原则和约束条件。

如上面删除文章的路由设计：

```
GET /posts/:postId/remove
```

restful 风格的设计：

```
DELETE /post/:postId
```

可以看出，restful 风格的 api 更直观且优雅。

更多阅读：

1. http://www.ruanyifeng.com/blog/2011/09/restful
2. http://www.ruanyifeng.com/blog/2014/05/restful_api.html
3. http://developer.51cto.com/art/200908/141825.htm
4. http://blog.jobbole.com/41233/

4.4.2 会话

由于 HTTP 协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是会话（Session）。关于 Session 的讲解网上有许多资料，这里不再赘述。参考：

1. http://justsee.iteye.com/blog/1570652
2. https://www.zhihu.com/question/19786827

cookie 与 session 的区别

1. cookie 存储在浏览器（有大小限制），session 存储在服务端（没有大小限制）
2. 通常 session 的实现是基于 cookie 的，即 session id 存储于 cookie 中

我们通过引入 express-session 中间件实现对会话的支持：

```
app.use(session(options))
```

session 中间件会在 req 上添加 session 对象，即 req.session 初始值为 `{}`，当我们登录后设置 `req.session.user = 用户信息`，返回浏览器的头信息中会带上 `set-cookie` 将 session id 写到浏览器 cookie 中，那么该用户下次请求时，通过带上来的 cookie 中的 session id 我们就可以查找到该用户，并将用户信息保存到 `req.session.user`。

4.4.3 页面通知

我们还需要这样一个功能：当我们操作成功时需要显示一个成功的通知，如登录成功跳转到主页时，需要显示一个 `登陆成功` 的通知；当我们操作失败时需要显示一个失败的通知，如注册时用户名被占用了，需要显示一个 `用户名已占用` 的通知。通知只显示一次，刷新后消失，我们可以通过 connect-flash 中间件实现这个功能。

connect-flash 是基于 session 实现的，它的原理很简单：设置初始值 `req.session.flash={}`，通过 `req.flash(name, value)` 设置这个对象下的字段和值，通过 `req.flash(name)` 获取这个对象下的值，同时删除这个字段。

express-session、connect-mongo 和 connect-flash 的区别与联系

1. `express-session`：会话（session）支持中间件
2. `connect-mongo`：将 session 存储于 mongodb，需结合 express-session 使用，我们也可以将 session 存储于 redis，如 connect-redis
3. `connect-flash`：基于 session 实现的用于通知功能的中间件，需结合 express-session 使用

4.4.4 权限控制

不管是论坛还是博客网站，我们没有登录的话只能浏览，登陆后才能发帖或写文章，即使登录了你也不能修改或删除其他人的文章，这就是权限控制。我们也来给博客添加权限控制，如何实现页面的权限控制呢？我们可以把用户状态的检查封装成一个中间件，在每个需要权限控制的路由加载该中间件，即可实现页面的权限控制。在 myblog 下新建 middlewares 文件夹，在该目录下新建 check.js，添加如下代码：

middlewares/check.js

```
module.exports = {
  checkLogin: function checkLogin(req, res, next) {
    if (!req.session.user) {
      req.flash('error', '未登录');
      return res.redirect('/signin');
    }
    next();
  },

  checkNotLogin: function checkNotLogin(req, res, next) {
    if (req.session.user) {
      req.flash('error', '已登录');
      return res.redirect('back');//返回之前的页面
    }
    next();
  }
};
```

可以看出：

1. `checkLogin`：当用户信息（`req.session.user`）不存在，即认为用户没有登录，则跳转到登录页，同时显示 `未登录` 的通知，用于需要用户登录才能操作的页面及接口
2. `checkNotLogin`：当用户信息（`req.session.user`）存在，即认为用户已经登录，则跳转到之前的页面，同时显示 `已登录` 的通知，如登录、注册页面及登录、注册的接口

最终我们创建以下路由文件：

routes/index.js

```
module.exports = function (app) {
  app.get('/', function (req, res) {
    res.redirect('/posts');
  });
  app.use('/signup', require('./signup'));
  app.use('/signin', require('./signin'));
  app.use('/signout', require('./signout'));
  app.use('/posts', require('./posts'));
};
```

routes/posts.js

```
var express = require('express');
var router = express.Router();

var checkLogin = require('../middlewares/check').checkLogin;

// GET /posts 所有用户或者特定用户的文章页
//   eg: GET /posts?author=xxx
router.get('/', function(req, res, next) {
  res.send(req.flash());
});

// POST /posts 发表一篇文章
router.post('/', checkLogin, function(req, res, next) {
  res.send(req.flash());
});

// GET /posts/create 发表文章页
router.get('/create', checkLogin, function(req, res, next) {
  res.send(req.flash());
});

// GET /posts/:postId 单独一篇的文章页
router.get('/:postId', function(req, res, next) {
  res.send(req.flash());
});

// GET /posts/:postId/edit 更新文章页
```

```
router.get('/:postId/edit', checkLogin, function(req, res, next)
 {
  res.send(req.flash());
});

// POST /posts/:postId/edit 更新一篇文章
router.post('/:postId/edit', checkLogin, function(req, res, next
) {
  res.send(req.flash());
});

// GET /posts/:postId/remove 删除一篇文章
router.get('/:postId/remove', checkLogin, function(req, res, nex
t) {
  res.send(req.flash());
});

// POST /posts/:postId/comment 创建一条留言
router.post('/:postId/comment', checkLogin, function(req, res, n
ext) {
  res.send(req.flash());
});

// GET /posts/:postId/comment/:commentId/remove 删除一条留言
router.get('/:postId/comment/:commentId/remove', checkLogin, fun
ction(req, res, next) {
  res.send(req.flash());
});

module.exports = router;
```

routes/signin.js

```
var express = require('express');
var router = express.Router();

var checkNotLogin = require('../middlewares/check').checkNotLogi
n;

// GET /signin 登录页
router.get('/', checkNotLogin, function(req, res, next) {
  res.send(req.flash());
});

// POST /signin 用户登录
router.post('/', checkNotLogin, function(req, res, next) {
  res.send(req.flash());
});

module.exports = router;
```

routes/signup.js

```
var express = require('express');
var router = express.Router();

var checkNotLogin = require('../middlewares/check').checkNotLogi
n;

// GET /signup 注册页
router.get('/', checkNotLogin, function(req, res, next) {
  res.send(req.flash());
});

// POST /signup 用户注册
router.post('/', checkNotLogin, function(req, res, next) {
  res.send(req.flash());
});

module.exports = router;
```

routes/signout.js

```
var express = require('express');
var router = express.Router();

var checkLogin = require('../middlewares/check').checkLogin;

// GET /signout 登出
router.get('/', checkLogin, function(req, res, next) {
  res.send(req.flash());
});

module.exports = router;
```

最后，修改 index.js 如下：

index.js

```
var path = require('path');
var express = require('express');
var session = require('express-session');
var MongoStore = require('connect-mongo')(session);
var flash = require('connect-flash');
var config = require('config-lite');
var routes = require('./routes');
var pkg = require('./package');

var app = express();

// 设置模板目录
app.set('views', path.join(__dirname, 'views'));
// 设置模板引擎为 ejs
app.set('view engine', 'ejs');

// 设置静态文件目录
app.use(express.static(path.join(__dirname, 'public')));
// session 中间件
app.use(session({
  name: config.session.key,// 设置 cookie 中保存 session id 的字段
```

```
名称
  secret: config.session.secret,// 通过设置 secret 来计算 hash 值并放在 cookie 中，使产生的 signedCookie 防篡改
  cookie: {
    maxAge: config.session.maxAge// 过期时间，过期后 cookie 中的 session id 自动删除
  },
  store: new MongoStore({// 将 session 存储到 mongodb
    url: config.mongodb// mongodb 地址
  })
}));
// flash 中间价，用来显示通知
app.use(flash());

// 路由
routes(app);

// 监听端口，启动程序
app.listen(config.port, function () {
  console.log(`${pkg.name} listening on port ${config.port}`);
});
```

> 注意：中间件的加载顺序很重要。如上面设置静态文件目录的中间件应该放到 routes(app) 之前加载，这样静态文件的请求就不会落到业务逻辑的路由里；flash 中间件应该放到 session 中间件之后加载，因为 flash 是基于 session 的。

运行 `supervisor --harmony index` 启动博客，访问以下地址查看效果：

1. http://localhost:3000/posts
2. http://localhost:3000/signout
3. http://localhost:3000/signup

上一节：4.3 配置文件

下一节：4.5 页面设计

我们使用 jQuery + Semantic-UI 实现前端页面的设计，最终效果图如下:

注册页

myblog

my first blog

用户名*

用户名

密码*

密码

重复密码*

重复密码

性别*

男

头像*

选择文件 未选择任何文件

个人简介*

注册

登录页

myblog

my first blog

用户名*

用户名

密码*

密码

登录

未登录时的主页（或用户页）



登录后的主页（或用户页）



发表文章页

myblog
my first blog

标题*

内容*

发布

编辑文章页

myblog
my first blog

标题*
hello, world

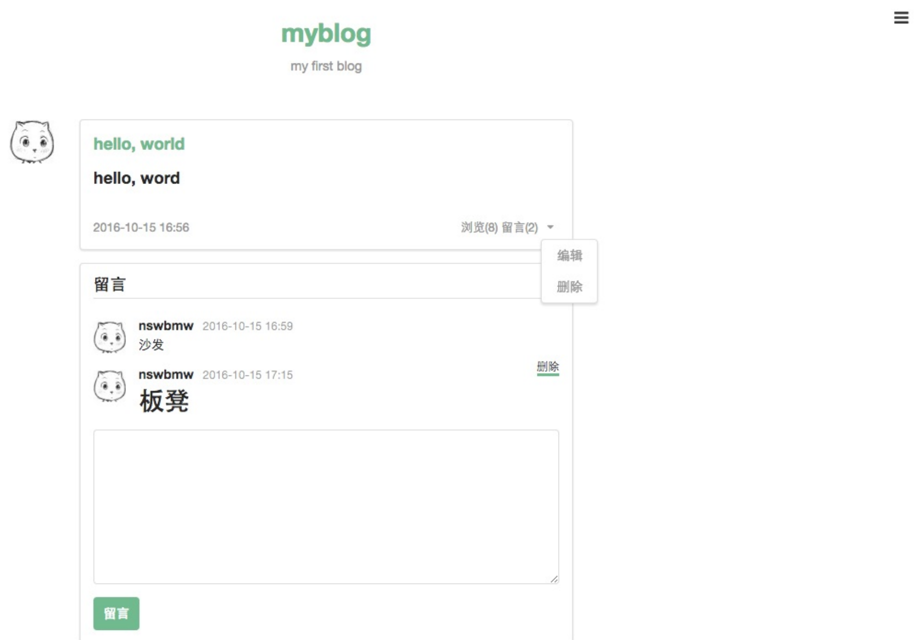
内容*
hello, word

发布

未登录时的文章页


myblog
my first blog
hello, world
hello, word
2016-10-15 16:56
浏览(7) 留言(2)
留言
nswbmw 2016-10-15 16:59
沙发
nswbmw 2016-10-15 17:15
板凳

登录后的文章页


myblog
my first blog
hello, world
hello, word
2016-10-15 16:56
浏览(8) 留言(2)
编辑
删除
留言
nswbmw 2016-10-15 16:59
沙发
nswbmw 2016-10-15 17:15
删除
板凳
留言

通知

myblog

my first blog

登录成功

Mongolass

```
const Mongolass = require('mongolass');
const Schema = Mongolass.Schema;
const mongolass = new Mongolass('mongodb://localhost:27017/t

const User = mongolass.model('User', {
  name: { type: 'string' },
  age: { type: 'number' }
});

User
  .insertOne({ name: 'nswbmw', age: 'wrong age' })
  .exec()
  .then(console.log)
  .catch(console.error);
```

2016-10-15 17:12 浏览(3) 留言(0)

hello, world

hello, word

2016-10-15 16:56 浏览(10) 留言(3)

myblog

my first blog

留言成功

hello, world

hello, word

2016-10-15 16:56 浏览(9) 留言(3)

留言

nswbmw 2016-10-15 16:59
沙发

nswbmw 2016-10-15 17:15
板凳

nswbmw 2016-10-15 20:01
哈哈

myblog

my first blog

用户名或密码错误

用户名*

用户名

密码*

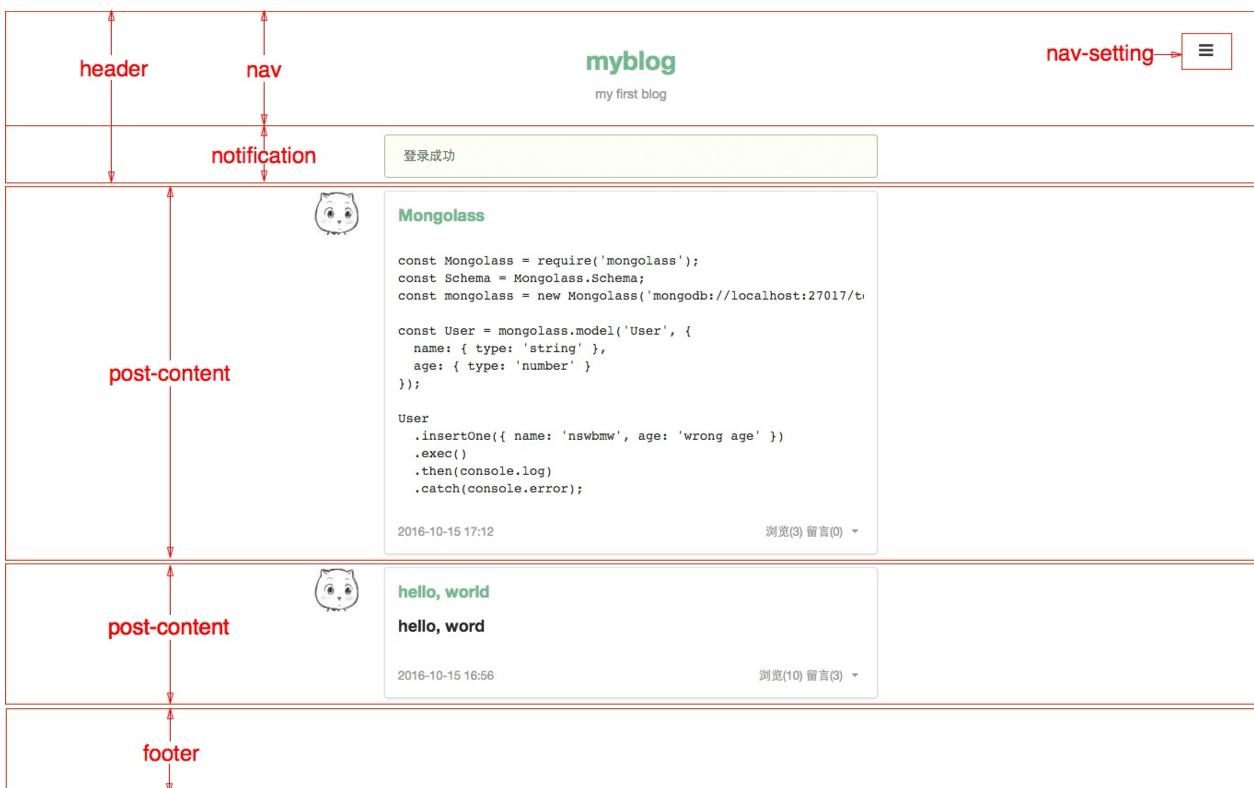
密码

登录

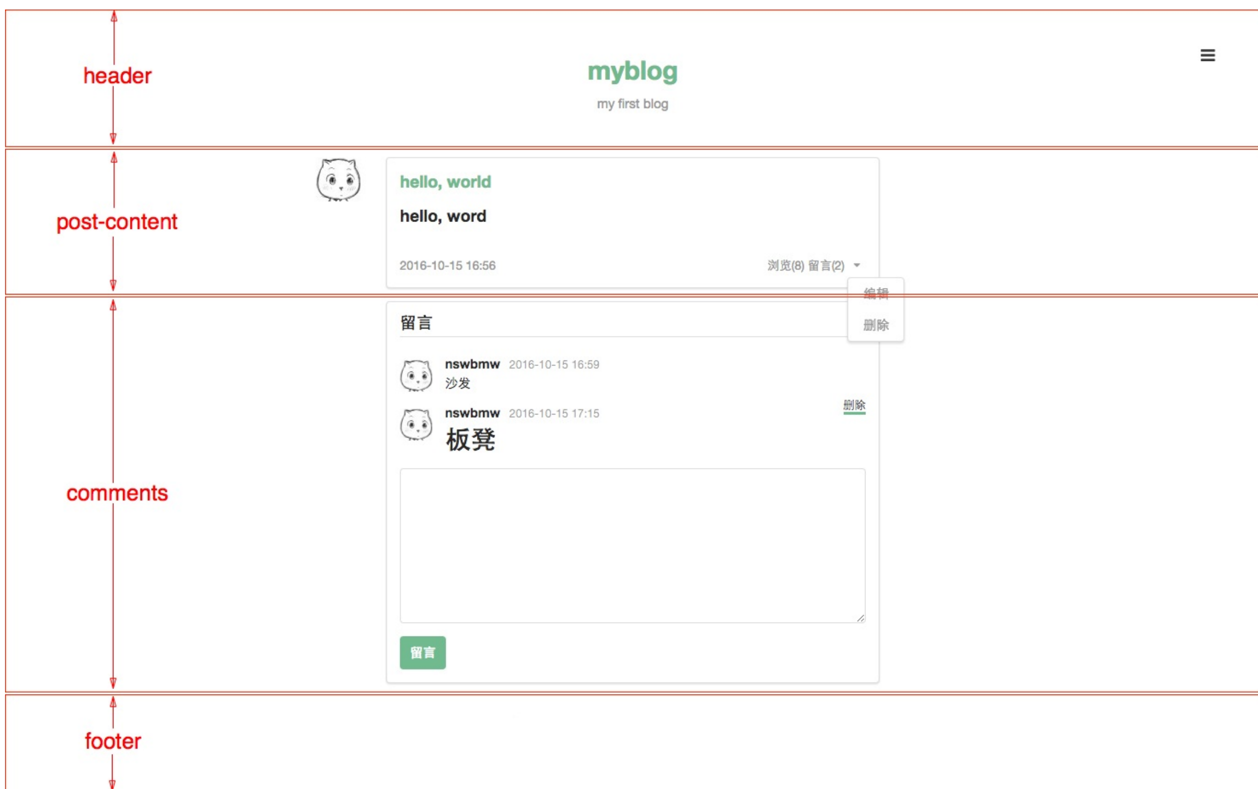
4.5.1 组件

前面提到过，我们可以将模板拆分成一些组件，然后使用 ejs 的 include 方法将组件组合起来进行渲染。我们将页面切分成以下组件：

主页



文章页



根据上面的组件切分图，我们创建以下样式及模板文件：

public/css/style.css

```
/* ---------- 全局样式 ---------- */

body {
  width: 1100px;
  height: 100%;
  margin: 0 auto;
  padding-top: 40px;
}

a:hover {
  border-bottom: 3px solid #4fc08d;
}

.button {
  background-color: #4fc08d !important;
  color: #fff !important;
}

.avatar {
```

```
  border-radius: 3px;
  width: 48px;
  height: 48px;
  float: right;
}

/* ---------- nav ---------- */

.nav {
  margin-bottom: 20px;
  color: #999;
  text-align: center;
}

.nav h1 {
  color: #4fc08d;
  display: inline-block;
  margin: 10px 0;
}

/* ---------- nav-setting ---------- */

.nav-setting {
  position: fixed;
  right: 30px;
  top: 35px;
  z-index: 999;
}

.nav-setting .ui.dropdown.button {
  padding: 10px 10px 0 10px;
  background-color: #fff !important;
}

.nav-setting .icon.bars {
  color: #000;
  font-size: 18px;
}

/* ---------- post-content ---------- */
```

```
.post-content h3 a {
  color: #4fc08d !important;
}

.post-content .tag {
  font-size: 13px;
  margin-right: 5px;
  color: #999;
}

.post-content .tag.right {
  float: right;
  margin-right: 0;
}

.post-content .tag.right a {
  color: #999;
}
```

views/header.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title><%= blog.title %></title>
    <link rel="stylesheet" href="//cdn.bootcss.com/semantic-ui/2
.1.8/semantic.min.css">
    <link rel="stylesheet" href="/css/style.css">
    <script src="//cdn.bootcss.com/jquery/1.11.3/jquery.min.js">
</script>
    <script src="//cdn.bootcss.com/semantic-ui/2.1.8/semantic.mi
n.js"></script>
  </head>
  <body>
  <%- include('components/nav') %>
  <%- include('components/nav-setting') %>
  <%- include('components/notification') %>
```

views/footer.ejs

```
  <script type="text/javascript">
    // 点击按钮弹出下拉框
    $('.ui.dropdown').dropdown();
    // 鼠标悬浮在头像上，弹出气泡提示框
    $('.post-content .avatar').popup({
      inline: true,
      position: 'bottom right',
      lastResort: 'bottom right',
    });
  </script>
  </body>
</html>
```

> 注意：上面 `<script></script>` 是 semantic-ui 操控页面控件的代码，一定要放到 footer.ejs 的 `</body>` 的前面，因为只有页面加载完后才能通过 JQuery 获取 DOM 元素。

在 views 目录下新建 components 目录用来存放组件，在该目录下创建以下文件：

views/components/nav.ejs

```
<div class="nav">
  <div class="ui grid">
    <div class="four wide column"></div>

    <div class="eight wide column">
      <a href="/posts"><h1><%= blog.title %></h1></a>
      <p><%= blog.description %></p>
    </div>
  </div>
</div>
```

views/components/nav-setting.ejs

```
<div class="nav-setting">
  <div class="ui buttons">
    <div class="ui floating dropdown button">
      <i class="icon bars"></i>
      <div class="menu">
        <% if (user) { %>
          <a class="item" href="/posts?author=<%= user._id %>">
个人主页</a>
          <div class="divider"></div>
          <a class="item" href="/posts/create">发表文章</a>
          <a class="item" href="/signout">登出</a>
        <% } else { %>
          <a class="item" href="/signin">登录</a>
          <a class="item" href="/signup">注册</a>
        <% } %>
      </div>
    </div>
  </div>
</div>
```

views/components/notification.ejs

```
<div class="ui grid">
  <div class="four wide column"></div>
  <div class="eight wide column">

  <% if (success) { %>
    <div class="ui success message">
      <p><%= success %></p>
    </div>
  <% } %>

  <% if (error) { %>
    <div class="ui error message">
      <p><%= error %></p>
    </div>
  <% } %>

  </div>
</div>
```

4.5.2 app.locals 和 res.locals

上面的模板中我们用到了 blog、user、success、error 变量，我们将 blog 变量挂载到 `app.locals` 下，将 user、success、error 挂载到 `res.locals` 下。为什么要这么做呢？ `app.locals` 和 `res.locals` 是什么？它们有什么区别？

express 中有两个对象可用于模板的渲染： `app.locals` 和 `res.locals` 。我们从 express 源码一探究竟：

express/lib/application.js

```
app.render = function render(name, options, callback) {
  ...
  var opts = options;
  var renderOptions = {};
  ...
  // merge app.locals
  merge(renderOptions, this.locals);

  // merge options._locals
  if (opts._locals) {
    merge(renderOptions, opts._locals);
  }

  // merge options
  merge(renderOptions, opts);
  ...
  tryRender(view, renderOptions, done);
};
```

express/lib/response.js

```
res.render = function render(view, options, callback) {
  var app = this.req.app;
  var opts = options || {};
  ...
  // merge res.locals
  opts._locals = self.locals;
  ...
  // render
  app.render(view, opts, done);
};
```

可以看出：在调用 `res.render` 的时候，express 合并（merge）了 3 处的结果后传入要渲染的模板，优先级：`res.render` 传入的对象> `res.locals` 对象 > `app.locals` 对象，所以 `app.locals` 和 `res.locals` 几乎没有区别，都用来渲染模板，使用上的区别在于：`app.locals` 上通常挂载常量信息（如博客名、描述、作者信息），`res.locals` 上通常挂载变量信息，即每次请求可能的值都不一样（如请求者信息，`res.locals.user = req.session.user`）。

修改 index.js，在 `routes(app);` 上一行添加如下代码：

```
// 设置模板全局常量
app.locals.blog = {
  title: pkg.name,
  description: pkg.description
};

// 添加模板必需的三个变量
app.use(function (req, res, next) {
  res.locals.user = req.session.user;
  res.locals.success = req.flash('success').toString();
  res.locals.error = req.flash('error').toString();
  next();
});
```

这样在调用 `res.render` 的时候就不用传入这四个变量了，express 为我们自动 merge 并传入了模板，所以我们可以在模板中直接使用这四个变量。

上一节：4.4 功能设计

下一节：4.6 连接数据库

我们使用 Mongolass 这个模块操作 mongodb 进行增删改查。在 myblog 下新建 lib 目录，在该目录下新建 mongo.js，添加如下代码：

lib/mongo.js

```
var config = require('config-lite');
var Mongolass = require('mongolass');
var mongolass = new Mongolass();
mongolass.connect(config.mongodb);
```

4.6.1 为什么使用 Mongolass

早期我使用官方的 mongodb（也叫 node-mongodb-native）库，后来也陆续尝试使用了许多其他 mongodb 的驱动库，Mongoose 是比较优秀的一个，使用 Mongoose 的时间也比较长。比较这两者，各有优缺点。

node-mongodb-native:

优点：

1. 简单。参照文档即可上手，没有 Mongoose 的 Schema 那些对新手不友好的东西。
2. 强大。毕竟是官方库，包含了所有且最新的 api，其他大部分的库都是在这个库的基础上改造的，包括 Mongoose。
3. 文档健全。

缺点：

1. 起初只支持 callback，会写出以下这种代码：

```
mongodb.open(function (err, db) {
if (err) {
return callback(err);
}
db.collection('users', function (err, collection) {
if (err) {
  return callback(err);
}
collection.find({ name: 'xxx' }, function (err, users) {
  if (err) {
    return callback(err);
  }
})
...
```

或者：

```
MongoClient.connect('mongodb://localhost:27017', function (err,
mongodb) {
  if (err) {
    return callback(err);
  }
  mongodb.db('test').collection('users').find({ name: 'xxx' }, f
unction (err, users) {
    if (err) {
      return callback(err);
    }
  })
  ...
```

现在支持 Promise 了，和 co 一起使用好很多。

1. 不支持文档校验。Mongoose 通过 Schema 支持文档校验，虽说 mongodb 是 no schema 的，但在生产环境中使用 Schema 有两点好处。一是对文档做校验，防止非正常情况下写入错误的数据到数据库，二是可以简化一些代码，如类型为 ObjectId 的字段查询或更新时可通过对应的字符串操作，不用每次包装成 ObjectId 对象。

Mongoose:

优点：

1. 封装了数据库的操作，给人的感觉是同步的，其实内部是异步的。如 mongoose 与 MongoDB 建立连接：

   ```
   var mongoose = require('mongoose');
   mongoose.connect('mongodb://localhost/test');
   var BlogModel = mongoose.model('Blog', { title: String, cont
   ent: String });
   BlogModel.find()
   ```

2. 支持 Promise。这个也无需多说，Promise 是未来趋势，可结合 co 使用，也可结合 async/await 使用。
3. 支持文档校验。如上所述。

缺点（个人观点）：

1. 功能多，复杂。Mongoose 很强大，有很多功能是比较鸡肋甚至可以去掉的，如果使用反而会影响代码的可读性。比如虚拟属性以及 schema 上定义 statics 方法和 instance 方法，可以把这些定义成外部方法，用到的时候调用即可。
2. 较弱的 plugin 系统。如：`schema.pre('save', function(next) {})` 和 `schema.post('find', function(next) {})`，只支持异步 `next()`，灵活性大打折扣。
3. 其他：对新手来说难以理解的 Schema、Model、Entry 之间的关系；容易混淆的 toJSON 和 toObject，以及有带有虚拟属性的情况；用和不用 exec 的情况以及直接用 then 的情况；返回的结果是 Mongoose 包装后的对象，在此对象上修改结果却无效等等。

Mongolass

Mongolass 保持了与 mongodb 一样的 api，又借鉴了许多 Mongoose 的优点，同时又保持了精简。

优点：

1. 支持 Promise。
2. 简单。参考 Mongolass 的 readme 即可上手，比 Mongoose 精简的多，本身代码也不多。

3. 可选的 Schema。Mongolass 中的 Schema （基于 another-json-schema）是可选的，并且只用来做文档校验。如果定义了 schema 并关联到某个 model，则插入、更新和覆盖等操作都会校验文档是否满足 schema，同时 schema 也会尝试格式化该字段，类似于 Mongoose，如定义了一个字段为 ObjectId 类型，也可以用 ObjectId 的字符串无缝使用一样。如果没有 schema，则用法跟原生 mongodb 库一样。
4. 简单却强大的插件系统。可以定义全局插件（对所有 model 生效），也可以定义某个 model 上的插件（只对该 model 生效）。Mongolass 插件的设计思路借鉴了中间件的概念（类似于 Koa），通过定义 `beforeXXX` 和 `afterXXX` （XXX为操作符首字母大写，如： `afterFind` ）函数实现，函数返回 yieldable 的对象即可，所以每个插件内可以做一些其他的 IO 操作。不同的插件顺序会有不同的结果，而且每个插件的输入输出都是 plain object，而非 Mongolass 包装后的对象，没有虚拟属性，无需调用 toJSON 或 toObject。Mongolass 中的 `.populate()` 就是一个内置的插件。
5. 详细的错误信息。用过 Mongoose 的人一定遇到过这样的错： `CastError: Cast to ObjectId failed for value "xxx" at path "_id"` 只知道一个期望是 ObjectId 的字段传入了非期望的值，通常很难定位出错的代码，即使定位到也得不到错误现场。得益于 another-json-schema，使用 Mongolass 在查询或者更新时，某个字段不匹配它定义的 schema 时（还没落到 mongodb）会给出详细的错误信息，如下所示： ``` const Mongolass = require('mongolass'); const mongolass = new Mongolass('mongodb://localhost:27017/test');

const User = mongolass.model('User', { name: { type: 'string' }, age: { type: 'number' } });

User .insertOne({ name: 'nswbmw', age: 'wrong age' }) .exec() .then(console.log) .catch(function (e) { console.error(e); console.error(e.stack); }); / *{ [Error: ($.age: "wrong age") ✖ (type: number)] validator: 'type', actual: 'wrong age', expected: { type: 'number' }, path: '$.age', schema: 'UserSchema', model: 'User', plugin: 'MongolassSchema', type: 'beforeInsertOne', args: [] } Error at Model.insertOne (/Users/nswbmw/Desktop/mongolass-demo/node_modules/mongolass/lib/query.js:107:16) at Object. (/Users/nswbmw/Desktop/mongolass-demo/app.js:10:4) at Module._compile (module.js:409:26) at Object.Module._extensions..js (module.js:416:10) at Module.load (module.js:343:32) at Function.Module._load (module.js:300:12) at Function.Module.runMain (module.js:441:10) at startup (node.js:139:18) at*

node.js:974:3 / `` 可以看出，错误的原因是在 insertOne 一条用户数据到用户表的时候，age 期望是一个 number 类型的值，而我们传入的字符串 wrong age`，然后从错误栈中可以快速定位到是 app.js 第 10 行代码抛出的错。

缺点：

1. schema 功能较弱，缺少如 required、default 功能。

上一节：4.5 页面设计

下一节：4.7 注册

4.7.1 用户模型设计

我们只存储用户的名称、密码（加密后的）、头像、性别和个人简介这几个字段，对应修改 lib/mongo.js，添加如下代码：

lib/mongo.js

```
exports.User = mongolass.model('User', {
  name: { type: 'string' },
  password: { type: 'string' },
  avatar: { type: 'string' },
  gender: { type: 'string', enum: ['m', 'f', 'x'] },
  bio: { type: 'string' }
});
exports.User.index({ name: 1 }, { unique: true }).exec();// 根据
用户名找到用户，用户名全局唯一
```

我们定义了用户表的 schema，生成并导出了 User 这个 model，同时设置了 name 的唯一索引，保证用户名是不重复的。

> 小提示：关于 Mongolass 的 schema 的用法，请查阅 another-json-schema。
>
> 小提示：Mongolass 中的 model 你可以认为相当于 mongodb 中的 collection，只不过添加了插件的功能。

4.7.2 注册页

首先，我们来完成注册。新建 views/signup.ejs，添加如下代码：

views/signup.ejs

```
<%- include('header') %>

<div class="ui grid">
  <div class="four wide column"></div>
  <div class="eight wide column">
    <form class="ui form segment" method="post" enctype="multipa
```

```
rt/form-data">
      <div class="field required">
        <label>用户名</label>
        <input placeholder="用户名" type="text" name="name">
      </div>
      <div class="field required">
        <label>密码</label>
        <input placeholder="密码" type="password" name="password"
>
      </div>
      <div class="field required">
        <label>重复密码</label>
        <input placeholder="重复密码" type="password" name="repas
sword">
      </div>
      <div class="field required">
        <label>性别</label>
        <select class="ui compact selection dropdown" name="gend
er">
          <option value="m">男</option>
          <option value="f">女</option>
          <option value="x">保密</option>
        </select>
      </div>
      <div class="field required">
        <label>头像</label>
        <input type="file" name="avatar">
      </div>
      <div class="field required">
        <label>个人简介</label>
        <textarea name="bio" rows="5" v-model="user.bio"></texta
rea>
      </div>
      <input type="submit" class="ui button fluid" value="注册">
    </div>
  </form>
</div>

<%- include('footer') %>
```

> 注意：form 表单要添加 `enctype="multipart/form-data"` 属性才能上传文件。

修改 routes/signup.js 中获取注册页的路由如下：

routes/signup.js

```
// GET /signup 注册页
router.get('/', checkNotLogin, function(req, res, next) {
  res.render('signup');
});
```

现在访问 `localhost:3000/signup` 看看效果吧。

4.7.3 注册与文件上传

我们使用 express-formidable 处理 form 表单（包括文件上传）。修改 index.js ，在 `app.use(flash());` 下一行添加如下代码：

index.js

```
// 处理表单及文件上传的中间件
app.use(require('express-formidable')({
  uploadDir: path.join(__dirname, 'public/img'),// 上传文件目录
  keepExtensions: true// 保留后缀
}));
```

新建 models/users.js，添加如下代码：

models/users.js

```
var User = require('../lib/mongo').User;

module.exports = {
  // 注册一个用户
  create: function create(user) {
    return User.create(user).exec();
  }
};
```

完善处理用户注册的路由，最终修改 routes/signup.js 如下：

routes/signup.js

```
var path = require('path');
var sha1 = require('sha1');
var express = require('express');
var router = express.Router();

var UserModel = require('../models/users');
var checkNotLogin = require('../middlewares/check').checkNotLogi
n;

// GET /signup 注册页
router.get('/', checkNotLogin, function(req, res, next) {
  res.render('signup');
});

// POST /signup 用户注册
router.post('/', checkNotLogin, function(req, res, next) {
  var name = req.fields.name;
  var gender = req.fields.gender;
  var bio = req.fields.bio;
  var avatar = req.files.avatar.path.split(path.sep).pop();
  var password = req.fields.password;
  var repassword = req.fields.repassword;

  // 校验参数
  try {
    if (!(name.length >= 1 && name.length <= 10)) {
```

```
      throw new Error('名字请限制在 1-10 个字符');
    }
    if (['m', 'f', 'x'].indexOf(gender) === -1) {
      throw new Error('性别只能是 m、f 或 x');
    }
    if (!(bio.length >= 1 && bio.length <= 30)) {
      throw new Error('个人简介请限制在 1-30 个字符');
    }
    if (!req.files.avatar.name) {
      throw new Error('缺少头像');
    }
    if (password.length < 6) {
      throw new Error('密码至少 6 个字符');
    }
    if (password !== repassword) {
      throw new Error('两次输入密码不一致');
    }
  } catch (e) {
    req.flash('error', e.message);
    return res.redirect('/signup');
  }

  // 明文密码加密
  password = sha1(password);

  // 待写入数据库的用户信息
  var user = {
    name: name,
    password: password,
    gender: gender,
    bio: bio,
    avatar: avatar
  };
  // 用户信息写入数据库
  UserModel.create(user)
    .then(function (result) {
      // 此 user 是插入 mongodb 后的值，包含 _id
      user = result.ops[0];
      // 将用户信息存入 session
      delete user.password;
```

```
        req.session.user = user;
        // 写入 flash
        req.flash('success', '注册成功');
        // 跳转到首页
        res.redirect('/posts');
      })
      .catch(function (e) {
        // 用户名被占用则跳回注册页，而不是错误页
        if (e.message.match('E11000 duplicate key')) {
          req.flash('error', '用户名已被占用');
          return res.redirect('/signup');
        }
        next(e);
      });
});

module.exports = router;
```

> 注意：我们使用 sha1 加密用户的密码，sha1 并不是一种十分安全的加密方式，实际开发中可以使用更安全的 bcrypt 或 scrypt 加密。

为了方便观察效果，我们先创建主页的模板。修改 routes/posts.js 中对应代码如下：

routes/posts.js

```
router.get('/', function(req, res, next) {
  res.render('posts');
});
```

新建 views/posts.ejs，添加如下代码：

views/posts.ejs

```
<%- include('header') %>
这是主页
<%- include('footer') %>
```

访问 `localhost:3000/signup` ，注册成功后如下所示：



上一节：4.6 连接数据库

下一节：4.8 登出与登录

4.8.1 登出

现在我们来完成登出的功能。修改 routes/signout.js 如下：

routes/signout.js

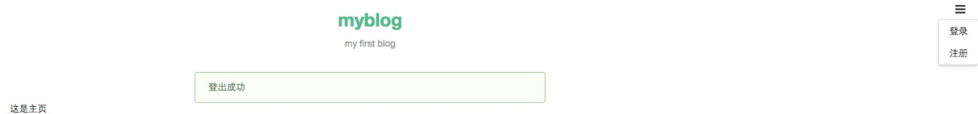
```
var express = require('express');
var router = express.Router();

var checkLogin = require('../middlewares/check').checkLogin;

// GET /signout 登出
router.get('/', checkLogin, function(req, res, next) {
  // 清空 session 中用户信息
  req.session.user = null;
  req.flash('success', '登出成功');
  // 登出成功后跳转到主页
  res.redirect('/posts');
});

module.exports = router;
```

此时点击右上角的 `登出` ，成功后如下图所示：



4.8.2 登录页

现在我们来完成登录页。修改 routes/signin.js 相应代码如下：

routes/signin.js

```
router.get('/', checkNotLogin, function(req, res, next) {
  res.render('signin');
});
```

新建 views/signin.ejs，添加如下代码：

views/signin.ejs

```
<%- include('header') %>

<div class="ui grid">
  <div class="four wide column"></div>
  <div class="eight wide column">
    <form class="ui form segment" method="post">
      <div class="field required">
        <label>用户名</label>
        <input placeholder="用户名" type="text" name="name">
      </div>
      <div class="field required">
        <label>密码</label>
        <input placeholder="密码" type="password" name="password"
>
      </div>
      <input type="submit" class="ui button fluid" value="登录">
    </div>
  </form>
</div>

<%- include('footer') %>
```

现在访问 `localhost:3000/signin` 试试吧。

4.8.3 登录

现在我们来完成登录的功能。修改 models/users.js 添加 `getUserByName` 方法用于通过用户名获取用户信息：

models/users.js

```
var User = require('../lib/mongo').User;

module.exports = {
  // 注册一个用户
  create: function create(user) {
    return User.create(user).exec();
  },

  // 通过用户名获取用户信息
  getUserByName: function getUserByName(name) {
    return User
      .findOne({ name: name })
      .addCreatedAt()
      .exec();
  }
};
```

这里我们使用了 `addCreatedAt` 自定义插件（通过 _id 生成时间戳），修改 lib/mongo.js，添加如下代码：

lib/mongo.js

```
var moment = require('moment');
var objectIdToTimestamp = require('objectid-to-timestamp');

// 根据 id 生成创建时间 created_at
mongolass.plugin('addCreatedAt', {
  afterFind: function (results) {
    results.forEach(function (item) {
      item.created_at = moment(objectIdToTimestamp(item._id)).format('YYYY-MM-DD HH:mm');
    });
    return results;
  },
  afterFindOne: function (result) {
    if (result) {
      result.created_at = moment(objectIdToTimestamp(result._id)).format('YYYY-MM-DD HH:mm');
    }
    return result;
  }
});
```

> 小提示：24 位长的 ObjectId 前 4 个字节是精确到秒的时间戳，所以我们没有额外的存创建时间（如: createdAt）的字段。ObjectId 生成规则：

0	1	2	3	4	5	6	7	8	9	10	11
时间戳				机器			PID		计数器		

修改 routes/signin.js 如下：

routes/signin.js

```
var sha1 = require('sha1');
var express = require('express');
var router = express.Router();

var UserModel = require('../models/users');
var checkNotLogin = require('../middlewares/check').checkNotLogin;

// GET /signin 登录页
```

```
router.get('/', checkNotLogin, function(req, res, next) {
  res.render('signin');
});

// POST /signin 用户登录
router.post('/', checkNotLogin, function(req, res, next) {
  var name = req.fields.name;
  var password = req.fields.password;

  UserModel.getUserByName(name)
    .then(function (user) {
      if (!user) {
        req.flash('error', '用户不存在');
        return res.redirect('back');
      }
      // 检查密码是否匹配
      if (sha1(password) !== user.password) {
        req.flash('error', '用户名或密码错误');
        return res.redirect('back');
      }
      req.flash('success', '登录成功');
      // 用户信息写入 session
      delete user.password;
      req.session.user = user;
      // 跳转到主页
      res.redirect('/posts');
    })
    .catch(next);
});

module.exports = router;
```

现在访问 `localhost:3000/signin` ，用刚才注册的账号登录，如下图所示：

上一节：4.7 注册

下一节：4.9 文章

4.9.1 文章模型设计

我们只存储文章的作者 id、标题、正文和点击量这几个字段，对应修改 lib/mongo.js，添加如下代码：

lib/mongo.js

```
exports.Post = mongolass.model('Post', {
  author: { type: Mongolass.Types.ObjectId },
  title: { type: 'string' },
  content: { type: 'string' },
  pv: { type: 'number' }
});
exports.Post.index({ author: 1, _id: -1 }).exec();// 按创建时间降序查看用户的文章列表
```

4.9.2 发表文章

现在我们来实现发表文章的功能。首先创建发表文章页，新建 views/create.ejs，添加如下代码：

views/create.ejs

```
<%- include('header') %>

<div class="ui grid">
  <div class="four wide column">
    <a class="avatar"
       href="/posts?author=<%= user._id %>"
       data-title="<%= user.name %> | <%= ({m: '男', f: '女', x:
'保密'})[user.gender] %>"
       data-content="<%= user.bio %>">
      <img class="avatar" src="/img/<%= user.avatar %>">
    </a>
  </div>

  <div class="eight wide column">
    <form class="ui form segment" method="post" action="/posts">
      <div class="field required">
        <label>标题</label>
        <input type="text" name="title">
      </div>
      <div class="field required">
        <label>内容</label>
        <textarea name="content" rows="15"></textarea>
      </div>
      <input type="submit" class="ui button" value="发布">
    </div>
  </form>
</div>

<%- include('footer') %>
```

新建 models/posts.js 用来存放与文章操作相关的代码：

models/posts.js

```
var Post = require('../lib/mongo').Post;

module.exports = {
  // 创建一篇文章
  create: function create(post) {
    return Post.create(post).exec();
  }
};
```

修改 routes/posts.js，在文件上方引入 PostModel：

routes/posts.js

```
var PostModel = require('../models/posts');
```

将：

```
// GET /posts/create 发表文章页
router.get('/create', checkLogin, function(req, res, next) {
  res.send(req.flash());
});

// POST /posts 发表一篇文章
router.post('/', checkLogin, function(req, res, next) {
  res.send(req.flash());
});
```

修改为：

```
// GET /posts/create 发表文章页
router.get('/create', checkLogin, function(req, res, next) {
  res.render('create');
});

// POST /posts 发表一篇文章
router.post('/', checkLogin, function(req, res, next) {
  var author = req.session.user._id;
```

```
  var title = req.fields.title;
  var content = req.fields.content;

  // 校验参数
  try {
    if (!title.length) {
      throw new Error('请填写标题');
    }
    if (!content.length) {
      throw new Error('请填写内容');
    }
  } catch (e) {
    req.flash('error', e.message);
    return res.redirect('back');
  }

  var post = {
    author: author,
    title: title,
    content: content,
    pv: 0
  };

  PostModel.create(post)
    .then(function (result) {
      // 此 post 是插入 mongodb 后的值，包含 _id
      post = result.ops[0];
      req.flash('success', '发表成功');
      // 发表成功后跳转到该文章页
      res.redirect(`/posts/${post._id}`);
    })
    .catch(next);
});
```

现在访问 `localhost:3000/posts/create` 发表篇文章试试吧，发表成功后跳转到了文章页但并没有任何内容，下面我们就来实现文章页及主页。

4.9.3 主页与文章页

现在我们来实现主页及文章页。修改 models/posts.js 如下：

models/posts.js

```
var marked = require('marked');
var Post = require('../lib/mongo').Post;

// 将 post 的 content 从 markdown 转换成 html
Post.plugin('contentToHtml', {
  afterFind: function (posts) {
    return posts.map(function (post) {
      post.content = marked(post.content);
      return post;
    });
  },
  afterFindOne: function (post) {
    if (post) {
      post.content = marked(post.content);
    }
    return post;
  }
});

module.exports = {
  // 创建一篇文章
  create: function create(post) {
    return Post.create(post).exec();
  },

  // 通过文章 id 获取一篇文章
  getPostById: function getPostById(postId) {
    return Post
      .findOne({ _id: postId })
      .populate({ path: 'author', model: 'User' })
      .addCreatedAt()
      .contentToHtml()
      .exec();
  },

  // 按创建时间降序获取所有用户文章或者某个特定用户的所有文章
```

```
  getPosts: function getPosts(author) {
    var query = {};
    if (author) {
      query.author = author;
    }
    return Post
      .find(query)
      .populate({ path: 'author', model: 'User' })
      .sort({ _id: -1 })
      .addCreatedAt()
      .contentToHtml()
      .exec();
  },

  // 通过文章 id 给 pv 加 1
  incPv: function incPv(postId) {
    return Post
      .update({ _id: postId }, { $inc: { pv: 1 } })
      .exec();
  }
};
```

需要讲解两点：

1. 我们使用了 markdown 解析文章的内容，所以在发表文章的时候可使用 markdown 语法（如插入链接、图片等等），关于 markdown 的使用请参考：Markdown 语法说明。
2. 我们在 PostModel 上注册了 `contentToHtml`，而 `addCreatedAt` 是在 lib/mongo.js 中 mongolass 上注册的。

接下来完成主页的模板，修改 views/posts.ejs 如下：

views/posts.ejs

```
<%- include('header') %>

<% posts.forEach(function (post) { %>
  <%- include('components/post-content', { post: post }) %>
<% }) %>

<%- include('footer') %>
```

新建 views/components/post-content.ejs 用来存放单篇文章的模板片段：

views/components/post-content.ejs

```
<div class="post-content">
  <div class="ui grid">
    <div class="four wide column">
      <a class="avatar"
         href="/posts?author=<%= post.author._id %>"
         data-title="<%= post.author.name %> | <%= ({m: '男', f:
 '女', x: '保密'})[post.author.gender] %>"
         data-content="<%= post.author.bio %>">
        <img class="avatar" src="/img/<%= post.author.avatar %>"
>
      </a>
    </div>

    <div class="eight wide column">
      <div class="ui segment">
        <h3><a href="/posts/<%= post._id %>"><%= post.title %></
a></h3>
        <pre><%- post.content %></pre>
        <div>
          <span class="tag"><%= post.created_at %></span>
          <span class="tag right">
            <span>浏览(<%= post.pv %>)</span>
            <span>留言(<%= post.commentsCount %>)</span>

            <% if (user && post.author._id && user._id.toString(
) === post.author._id.toString()) { %>
              <div class="ui inline dropdown">
```

```
                <div class="text"></div>
                <i class="dropdown icon"></i>
                <div class="menu">
                  <div class="item"><a href="/posts/<%= post._id
 %>/edit">编辑</a></div>
                  <div class="item"><a href="/posts/<%= post._id
 %>/remove">删除</a></div>
                </div>
              </div>
            <% } %>

          </span>
        </div>
      </div>
    </div>
  </div>
</div>
```

> 注意：我们用了 `<%- post.content %>`，而不是 `<%= post.content %>`，因为 post.content 是 markdown 转换成的 html 字符串。

修改 routes/posts.js，将：

routes/posts.js

```
router.get('/', function(req, res, next) {
  res.render('posts');
});
```

修改为：

```
router.get('/', function(req, res, next) {
  var author = req.query.author;

  PostModel.getPosts(author)
    .then(function (posts) {
      res.render('posts', {
        posts: posts
      });
    })
    .catch(next);
});
```

> 注意：主页与用户页通过 url 中的 author 区分。

现在完成了主页与用户页，访问 `http://localhost:3000/posts` 试试吧，尝试点击用户的头像看看效果。

接下来完成文章页。新建 views/post.ejs，添加如下代码：

views/post.ejs

```
<%- include('header') %>
<%- include('components/post-content') %>
<%- include('footer') %>
```

打开 routes/posts.js，将：

routes/posts.js

```
// GET /posts/:postId 单独一篇的文章页
router.get('/:postId', function(req, res, next) {
  res.send(req.flash());
});
```

修改为：

```
// GET /posts/:postId 单独一篇的文章页
router.get('/:postId', function(req, res, next) {
  var postId = req.params.postId;

  Promise.all([
    PostModel.getPostById(postId),// 获取文章信息
    PostModel.incPv(postId)// pv 加 1
  ])
  .then(function (result) {
    var post = result[0];
    if (!post) {
      throw new Error('该文章不存在');
    }

    res.render('post', {
      post: post
    });
  })
  .catch(next);
});
```

现在刷新浏览器，点击文章的标题看看浏览器地址的变化吧。

4.9.4 编辑与删除文章

现在我们来完成编辑与删除文章的功能。修改 models/posts.js，在 module.exports 对象上添加如下 3 个方法：

models/posts.js

```
// 通过文章 id 获取一篇原生文章（编辑文章）
getRawPostById: function getRawPostById(postId) {
  return Post
    .findOne({ _id: postId })
    .populate({ path: 'author', model: 'User' })
    .exec();
},

// 通过用户 id 和文章 id 更新一篇文章
updatePostById: function updatePostById(postId, author, data) {
  return Post.update({ author: author, _id: postId }, { $set: da
ta }).exec();
},

// 通过用户 id 和文章 id 删除一篇文章
delPostById: function delPostById(postId, author) {
  return Post.remove({ author: author, _id: postId}).exec();
}
```

> 注意：不要忘了在适当位置添加逗号，如 incPv 的结束大括号后。
>
> 注意：我们通过新函数 `getRawPostById` 用来获取文章原生的内容，而不是用 `getPostById` 返回将 markdown 转换成 html 后的内容。

新建编辑文章页 views/edit.ejs，添加如下代码：

views/edit.ejs

```
<%- include('header') %>

<div class="ui grid">
  <div class="four wide column">
    <a class="avatar"
       href="/posts?author=<%= user._id %>"
       data-title="<%= user.name %> | <%= ({m: '男', f: '女', x:
'保密'})[user.gender] %>"
       data-content="<%= user.bio %>">
      <img class="avatar" src="/img/<%= user.avatar %>">
    </a>
  </div>

  <div class="eight wide column">
    <form class="ui form segment" method="post" action="/posts/<
%= post._id %>/edit">
      <div class="field required">
        <label>标题</label>
        <input type="text" name="title" value="<%= post.title %>
">
      </div>
      <div class="field required">
        <label>内容</label>
        <textarea name="content" rows="15"><%= post.content %></
textarea>
      </div>
      <input type="submit" class="ui button" value="发布">
    </div>
  </form>
</div>

<%- include('footer') %>
```

修改 routes/posts.js，将：

routes/posts.js

```
// GET /posts/:postId/edit 更新文章页
router.get('/:postId/edit', checkLogin, function(req, res, next)
 {
  res.send(req.flash());
});

// POST /posts/:postId/edit 更新一篇文章
router.post('/:postId/edit', checkLogin, function(req, res, next
) {
  res.send(req.flash());
});

// GET /posts/:postId/remove 删除一篇文章
router.get('/:postId/remove', checkLogin, function(req, res, nex
t) {
  res.send(req.flash());
});
```

修改为：

```
// GET /posts/:postId/edit 更新文章页
router.get('/:postId/edit', checkLogin, function(req, res, next)
 {
  var postId = req.params.postId;
  var author = req.session.user._id;

  PostModel.getRawPostById(postId)
    .then(function (post) {
      if (!post) {
        throw new Error('该文章不存在');
      }
      if (author.toString() !== post.author._id.toString()) {
        throw new Error('权限不足');
      }
      res.render('edit', {
        post: post
      });
    })
    .catch(next);
```

```
});

// POST /posts/:postId/edit 更新一篇文章
router.post('/:postId/edit', checkLogin, function(req, res, next
) {
  var postId = req.params.postId;
  var author = req.session.user._id;
  var title = req.fields.title;
  var content = req.fields.content;

  PostModel.updatePostById(postId, author, { title: title, conte
nt: content })
    .then(function () {
      req.flash('success', '编辑文章成功');
      // 编辑成功后跳转到上一页
      res.redirect(`/posts/${postId}`);
    })
    .catch(next);
});

// GET /posts/:postId/remove 删除一篇文章
router.get('/:postId/remove', checkLogin, function(req, res, nex
t) {
  var postId = req.params.postId;
  var author = req.session.user._id;

  PostModel.delPostById(postId, author)
    .then(function () {
      req.flash('success', '删除文章成功');
      // 删除成功后跳转到主页
      res.redirect('/posts');
    })
    .catch(next);
});
```

现在刷新主页，点击文章右下角的小三角，编辑文章和删除文章试试吧。

上一节：4.8 登出与登录

下一节：4.10 留言

4.10.1 留言模型设计

我们只需要留言的作者 id、留言内容和关联的文章 id 这几个字段，修改 lib/mongo.js，添加如下代码：

lib/mongo.js

```
exports.Comment = mongolass.model('Comment', {
  author: { type: Mongolass.Types.ObjectId },
  content: { type: 'string' },
  postId: { type: Mongolass.Types.ObjectId }
});
exports.Post.index({ postId: 1, _id: 1 }).exec();// 通过文章 id 获取该文章下所有留言，按留言创建时间升序
exports.Post.index({ author: 1, _id: 1 }).exec();// 通过用户 id 和留言 id 删除一个留言
```

4.10.2 显示留言

在实现留言功能之前，我们先让文章页可以显示留言列表。首先创建留言的模板，新建 views/components/comments.ejs，添加如下代码：

views/components/comments.ejs

```
<div class="ui grid">
  <div class="four wide column"></div>
  <div class="eight wide column">
    <div class="ui segment">
      <div class="ui minimal comments">
        <h3 class="ui dividing header">留言</h3>

        <% comments.forEach(function (comment) { %>
          <div class="comment">
            <span class="avatar">
              <img src="/img/<%= comment.author.avatar %>">
            </span>
            <div class="content">
```

```
              <a class="author" href="/posts?author=<%= comment.
author._id %>"><%= comment.author.name %></a>
              <div class="metadata">
                <span class="date"><%= comment.created_at %></sp
an>
              </div>
              <div class="text"><%- comment.content %></div>

              <% if (user && comment.author._id && user._id.toSt
ring() === comment.author._id.toString()) { %>
                <div class="actions">
                  <a class="reply" href="/posts/<%= post._id %>/
comment/<%= comment._id %>/remove">删除</a>
                </div>
              <% } %>
            </div>
          </div>
        <% }) %>

        <% if (user) { %>
          <form class="ui reply form" method="post" action="/pos
ts/<%= post._id %>/comment">
            <div class="field">
              <textarea name="content"></textarea>
            </div>
            <input type="submit" class="ui icon button" value="
留言" />
          </form>
        <% } %>

      </div>
    </div>
  </div>
</div>
```

在文章页引入留言的模板片段，修改 views/post.ejs 为：

views/post.ejs

```
<%- include('header') %>

<%- include('components/post-content') %>
<%- include('components/comments') %>

<%- include('footer') %>
```

新建 models/comments.js，添加如下代码：

models/comments.js

```
var marked = require('marked');
var Comment = require('../lib/mongo').Comment;

// 将 comment 的 content 从 markdown 转换成 html
Comment.plugin('contentToHtml', {
  afterFind: function (comments) {
    return comments.map(function (comment) {
      comment.content = marked(comment.content);
      return comment;
    });
  }
});

module.exports = {
  // 创建一个留言
  create: function create(comment) {
    return Comment.create(comment).exec();
  },

  // 通过用户 id 和留言 id 删除一个留言
  delCommentById: function delCommentById(commentId, author) {
    return Comment.remove({ author: author, _id: commentId }).ex
ec();
  },

  // 通过文章 id 获取该文章下所有留言，按留言创建时间升序
  getComments: function getComments(postId) {
    return Comment
```

```
      .find({ postId: postId })
      .populate({ path: 'author', model: 'User' })
      .sort({ _id: 1 })
      .addCreatedAt()
      .contentToHtml()
      .exec();
  },

  // 通过文章 id 获取该文章下留言数
  getCommentsCount: function getCommentsCount(postId) {
    return Comment.count({ postId: postId }).exec();
  }
};
```

> 小提示：我们让留言也支持了 markdown。
>
> 注意：其实通过 commentId 就可以唯一确定并删除一条留言，添加 author 的限制是为了防止用户删除他人的留言。

修改 models/posts.js，在：

models/posts.js

```
var Post = require('../lib/mongo').Post;
```

下添加如下代码：

```
var CommentModel = require('./comments');

// 给 post 添加留言数 commentsCount
Post.plugin('addCommentsCount', {
  afterFind: function (posts) {
    return Promise.all(posts.map(function (post) {
      return CommentModel.getCommentsCount(post._id).then(functi
on (commentsCount) {
        post.commentsCount = commentsCount;
        return post;
      });
    }));
  },
  afterFindOne: function (post) {
    if (post) {
      return CommentModel.getCommentsCount(post._id).then(functi
on (count) {
        post.commentsCount = count;
        return post;
      });
    }
    return post;
  }
});
```

在 PostModel 上注册了 `addCommentsCount` 用来给每篇文章添加留言数 `commentsCount`，在 `getPostById` 和 `getPosts` 方法里的：

```
.addCreatedAt()
```

下添加：

```
.addCommentsCount()
```

这样主页和文章页的文章就可以正常显示留言数了。

> 小提示：虽然目前看起来使用 Mongolass 自定义插件并不能节省代码，反而使代码变多了。Mongolass 插件真正的优势在于：在项目非常庞大时，可通过自定义的插件随意组合（及顺序）实现不同的输出，如上面的 `getPostById` 需要将取出 markdown 转换成 html，则使用 `.contentToHtml()` ，否则像 `getRawPostById` 则不使用。

修改 routes/posts.js，在：

routes/posts.js

```
var PostModel = require('../models/posts');
```

下引入 CommentModel:

```
var CommentModel = require('../models/comments');
```

在文章页传入留言列表，将：

```
// GET /posts/:postId 单独一篇的文章页
router.get('/:postId', function(req, res, next) {
  var postId = req.params.postId;

  Promise.all([
    PostModel.getPostById(postId),// 获取文章信息
    PostModel.incPv(postId)// pv 加 1
  ])
  .then(function (result) {
    var post = result[0];
    if (!post) {
      throw new Error('该文章不存在');
    }

    res.render('post', {
      post: post
    });
  })
  .catch(next);
});
```

修改为：

```
// GET /posts/:postId 单独一篇的文章页
router.get('/:postId', function(req, res, next) {
  var postId = req.params.postId;

  Promise.all([
    PostModel.getPostById(postId),// 获取文章信息
    CommentModel.getComments(postId),// 获取该文章所有留言
    PostModel.incPv(postId)// pv 加 1
  ])
  .then(function (result) {
    var post = result[0];
    var comments = result[1];
    if (!post) {
      throw new Error('该文章不存在');
    }

    res.render('post', {
      post: post,
      comments: comments
    });
  })
  .catch(next);
});
```

现在刷新文章页试试吧。

4.10.3 发表与删除留言

现在我们来实现发表与删除留言的功能。修改 routes/posts.js，将：

routes/posts.js

```
// POST /posts/:postId/comment 创建一条留言
router.post('/:postId/comment', checkLogin, function(req, res, next) {
  res.send(req.flash());
});

// GET /posts/:postId/comment/:commentId/remove 删除一条留言
router.get('/:postId/comment/:commentId/remove', checkLogin, function(req, res, next) {
  res.send(req.flash());
});
```

修改为：

```
// POST /posts/:postId/comment 创建一条留言
router.post('/:postId/comment', checkLogin, function(req, res, n
ext) {
  var author = req.session.user._id;
  var postId = req.params.postId;
  var content = req.fields.content;
  var comment = {
    author: author,
    postId: postId,
    content: content
  };

  CommentModel.create(comment)
    .then(function () {
      req.flash('success', '留言成功');
      // 留言成功后跳转到上一页
      res.redirect('back');
    })
    .catch(next);
});

// GET /posts/:postId/comment/:commentId/remove 删除一条留言
router.get('/:postId/comment/:commentId/remove', checkLogin, fun
ction(req, res, next) {
  var commentId = req.params.commentId;
  var author = req.session.user._id;

  CommentModel.delCommentById(commentId, author)
    .then(function () {
      req.flash('success', '删除留言成功');
      // 删除成功后跳转到上一页
      res.redirect('back');
    })
    .catch(next);
});
```

上一节：4.9 文章

下一节：4.11 404页面

现在访问一个不存在的地址，如：`http://localhost:3000/haha` 页面会显示：

```
Cannot GET /haha
```

我们来自定义 404 页面。修改 routes/index.js，在：

routes/index.js

```
app.use('/posts', require('./posts'));
```

下添加如下代码：

```
// 404 page
app.use(function (req, res) {
  if (!res.headersSent) {
    res.render('404');
  }
});
```

新建 views/404.ejs，添加如下代码：

views/404.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title><%= blog.title %></title>
    <script type="text/javascript" src="http://www.qq.com/404/se
arch_children.js" charset="utf-8"></script>
  </head>
  <body></body>
</html>
```

这里我们只为了演示 express 中处理 404 的情况，用了腾讯公益的 404 页面。

上一节：4.10 留言

下一节：4.12 错误页面

前面讲到 express 有一个内置的错误处理逻辑，如果程序出错，会直接将错误栈返回并显示到页面上。现在我们来自己写一个错误页面，新建 views/error.ejs，添加如下代码：

views/error.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title><%= blog.title %></title>
    <link rel="stylesheet" href="/css/style.css">
  </head>
  <body>
    <h2><%= error.message %></h2>
    <p><%= error.stack %></p>
  </body>
</html>
```

修改 index.js，在 `app.listen` 上一行添加如下代码：

index.js

```
// error page
app.use(function (err, req, res, next) {
  res.render('error', {
    error: err
  });
});
```

上一节：4.11 404页面

下一节：4.13 日志

现在我们来实现日志功能，日志分为正常请求的日志和错误请求的日志，这两种日志都打印到终端并写入文件。

4.13.1 winston 和 express-winston

我们使用 winston 和 express-winston 记录日志。新建 logs 目录存放日志文件，修改 index.js，在：

index.js

```
var pkg = require('./package');
```

下引入所需模块：

```
var winston = require('winston');
var expressWinston = require('express-winston');
```

将：

```
// 路由
routes(app);
```

修改为：

```
// 正常请求的日志
app.use(expressWinston.logger({
  transports: [
    new (winston.transports.Console)({
      json: true,
      colorize: true
    }),
    new winston.transports.File({
      filename: 'logs/success.log'
    })
  ]
}));
// 路由
routes(app);
// 错误请求的日志
app.use(expressWinston.errorLogger({
  transports: [
    new winston.transports.Console({
      json: true,
      colorize: true
    }),
    new winston.transports.File({
      filename: 'logs/error.log'
    })
  ]
}));
```

可以看出：我们将正常请求的日志打印到终端并写入了 `logs/success.log`，将错误请求的日志打印到终端并写入了 `logs/error.log`。需要注意的是：记录正常请求日志的中间件要放到 `routes(app)` 之前，记录错误请求日志的中间件要放到 `routes(app)` 之后。

4.13.2 .gitignore

如果我们想把项目托管到 git 服务器上（如: GitHub），而不想把线上配置、本地调试的 logs 以及 node_modules 添加到 git 的版本控制中，这个时候就需要 .gitignore 文件了，git 会读取 .gitignore 并忽略这些文件。在 myblog 下新建

.gitignore 文件，添加如下配置：

.gitignore

```
config/*
!config/default.*
logs
npm-debug.log
node_modules
coverage
```

需要注意的是，通过设置：

```
config/*
!config/default.*
```

这样只有 config/default.js 会加入 git 的版本控制，而 config 目录下的其他配置文件则会被忽略。

然后在 public/img 目录下创建 .gitignore：

```
# Ignore everything in this directory
*
# Except this file
!.gitignore
```

这样 git 会忽略 public/img 目录下所有上传的头像，而不忽略 public/img 目录。

上一节：4.12 错误页面

下一节：4.14 测试

4.14.1 mocha 和 supertest

mocha 和 suptertest 是常用的测试组合，通常用来测试 restful 的 api 接口，这里我们也可以用来测试我们的博客应用。在 myblog 下新建 test 文件夹存放测试文件，以注册为例讲解 mocha 和 supertest 的用法。首先安装所需模块：

```
npm i mocha supertest --save
```

修改 package.json，将：

package.json

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
}
```

修改为：

```
"scripts": {
  "test": "mocha --harmony test"
}
```

指定执行 test 目录的测试。修改 index.js，将：

index.js

```
// 监听端口，启动程序
app.listen(config.port, function () {
  console.log(`${pkg.name} listening on port ${config.port}`);
});
```

修改为:

```
if (module.parent) {
  module.exports = app;
} else {
  // 监听端口，启动程序
  app.listen(config.port, function () {
    console.log(`${pkg.name} listening on port ${config.port}`);
  });
}
```

这样做可以实现：直接启动 index.js 则会监听端口启动程序，如果 index.js 被 require 了，则导出 app，通常用于测试。

找一张图片用于测试上传头像，放到 test 目录下，如 avatar.png。新建 test/signup.js，添加如下测试代码：

test/signup.js

```
var path = require('path');
var assert = require('assert');
var request = require('supertest');
var app = require('../index');
var User = require('../lib/mongo').User;

describe('signup', function() {
  describe('POST /signup', function() {
    var agent = request.agent(app);//persist cookie when redirec
t
    beforeEach(function (done) {
      // 创建一个用户
      User.create({
        name: 'aaa',
        password: '123456',
        avatar: '',
        gender: 'x',
        bio: ''
      })
      .exec()
      .then(function () {
        done();
```

```
    })
    .catch(done);
  });

  afterEach(function (done) {
    // 清空 users 表
    User.remove({})
      .exec()
      .then(function () {
        done();
      })
      .catch(done);
  });

  // 名户名错误的情况
  it('wrong name', function(done) {
    agent
      .post('/signup')
      .type('form')
      .attach('avatar', path.join(__dirname, 'avatar.png'))
      .field({ name: '' })
      .redirects()
      .end(function(err, res) {
        if (err) return done(err);
        assert(res.text.match(/名字请限制在 1-10 个字符/));
        done();
      });
  });

  // 性别错误的情况
  it('wrong gender', function(done) {
    agent
      .post('/signup')
      .type('form')
      .attach('avatar', path.join(__dirname, 'avatar.png'))
      .field({ name: 'nswbmw', gender: 'a' })
      .redirects()
      .end(function(err, res) {
        if (err) return done(err);
        assert(res.text.match(/性别只能是 m、f 或 x/));
```

```
          done();
        });
    });
    // 其余的参数测试自行补充
    // 用户名被占用的情况
    it(‘duplicate name', function(done) {
      agent
        .post('/signup')
        .type('form')
        .attach('avatar', path.join(__dirname, 'avatar.png'))
        .field({ name: 'aaa', gender: 'm', bio: 'noder', passwor
d: '123456', repassword: '123456' })
        .redirects()
        .end(function(err, res) {
          if (err) return done(err);
          assert(res.text.match(/用户名已被占用/));
          done();
        });
    });

    // 注册成功的情况
    it('success', function(done) {
      agent
        .post('/signup')
        .type('form')
        .attach('avatar', path.join(__dirname, 'avatar.png'))
        .field({ name: 'nswbmw', gender: 'm', bio: 'noder', pass
word: '123456', repassword: '123456' })
        .redirects()
        .end(function(err, res) {
          if (err) return done(err);
          assert(res.text.match(/注册成功/));
          done();
        });
    });
  });
});
```

运行 `npm test` 看看效果吧，其余的测试请读者自行完成。

4.14.2 测试覆盖率

我们写测试肯定想覆盖所有的情况（包括各种出错的情况及正确时的情况），但光靠想需要写哪些测试是不行的，总也会有疏漏，最简单的办法就是可以直观的看出测试是否覆盖了所有的代码，这就是测试覆盖率，即被测试覆盖到的代码行数占总代码行数的比例。

> 注意：即使测试覆盖率达到 100% 也不能说明你的测试覆盖了所有的情况，只能说明基本覆盖了所有的情况。

istanbul 是一个常用的生成测试覆盖率的库，它会将测试的结果报告生成 html 页面，并放到项目根目录的 coverage 目录下。首先安装 istanbul:

```
npm i istanbul --save-dev
```

配置 istanbul 很简单，将 package.json 中：

package.json

```
"scripts": {
  "test": "mocha --harmony test"
}
```

修改为：

```
"scripts": {
  "test": "node --harmony ./node_modules/.bin/istanbul cover ./node_modules/.bin/_mocha"
}
```

> 注意：如果 Windows 下报错，尝试修改为：
>
> ```
> "scripts": {
>   "test": "node --harmony ./node_modules/istanbul/lib/cli.js
>  cover ./node_modules/mocha/bin/_mocha"
> }
> ```
>
> 见 #201.

即可将 mocha 和 istanbul 结合使用，终端会打印：

```
=============================== Coverage summary ===============================
Statements   : 57.2% ( 151/264 )
Branches     : 35.42% ( 17/48 )
Functions    : 33.9% ( 20/59 )
Lines        : 57.2% ( 151/264 )
================================================================================
```

打开 myblog/coverage/lcov-report/index.html，如下所示：

/

57.2% Statements 151/264 35.42% Branches 17/48 33.9% Functions 20/59 57.2% Lines 151/264

File		Statements		Branches		Functions		Lines	
myblog/		90.63%	29/32	50%	1/2	33.33%	1/3	90.63%	29/32
myblog/config/		100%	1/1	100%	0/0	100%	0/0	100%	1/1
myblog/lib/		85%	17/20	0%	0/2	66.67%	2/3	85%	17/20
myblog/middlewares/		33.33%	3/9	25%	1/4	50%	1/2	33.33%	3/9
myblog/models/		52.17%	24/46	16.67%	1/6	34.78%	8/23	52.17%	24/46
myblog/routes/		49.36%	77/156	41.18%	14/34	28.57%	8/28	49.36%	77/156

可以点进去查看某个代码文件具体的覆盖率，如下所示：

all files / myblog/routes/ signup.js

88.37% Statements 38/43 **72.22%** Branches 13/18 **100%** Functions 4/4 **88.37%** Lines 38/43

```
1  1×  var path = require('path');
2  1×  var sha1 = require('sha1');
3  1×  var express = require('express');
4  1×  var router = express.Router();
5
6  1×  var UserModel = require('../models/users');
7  1×  var checkNotLogin = require('../middlewares/check').checkNotLogin;
8
9      // GET /signup 注册页
10 1×  router.get('/', checkNotLogin, function(req, res, next) {
11 3×    res.render('signup');
12     });
13
14     // POST /signup 用户注册
15 1×  router.post('/', checkNotLogin, function(req, res, next) {
16 4×    var name = req.fields.name;
17 4×    var gender = req.fields.gender;
18 4×    var bio = req.fields.bio;
19 4×    var avatar = req.files.avatar.path.split(path.sep).pop();
20 4×    var password = req.fields.password;
21 4×    var repassword = req.fields.repassword;
22
23       // 校验参数
24 4×    try {
25 4×      if (!(name.length >= 1 && name.length <= 10)) {
26 1×        throw new Error('名字请限制在 1-10 个字符');
27         }
28 3×      if (['m', 'f', 'x'].indexOf(gender) === -1) {
29 1×        throw new Error('性别只能是 m、f 或 x');
30         }
31 2×      I if (!(bio.length >= 1 && bio.length <= 30)) {
32           throw new Error('个人简介请限制在 1-30 个字符');
33         }
34 2×      I if (!req.files.avatar.name) {
35           throw new Error('缺少头像');
36         }
37 2×      I if (password < 6) {
38           throw new Error('密码至少 6 个字符');
39         }
40 2×      I if (password !== repassword) {
41           throw new Error('两次输入密码不一致');
42         }
43       } catch (e) {
44 2×      req.flash('error', e.message);
45 2×      return res.redirect('/signup');
46       }
47
48       // 明文密码加密
49 2×    password = sha1(password);
50
51       // 待写入数据库的用户信息
52 2×    var user = {
53         name: name,
54         password: password,
55         gender: gender,
```

红色的行表示测试没有覆盖到，因为我们只写了 name 和 gender 的测试。

上一节：4.13 日志

下一节：4.15 部署

4.15.1 申请 MLab

MLab (前身是 MongoLab) 是一个 mongodb 云数据库提供商，我们可以选择 500MB 空间的免费套餐用来测试。注册成功后，点击右上角的 `Create New` 创建一个数据库（如: myblog），成功后点击进入到该数据库详情页，注意页面中有一行黄色的警告：

```
A database user is required to connect to this database. To crea
te one now, visit the 'Users' tab and click the 'Add database us
er' button.
```

每个数据库至少需要一个 user，所以我们点击 Users 下的 `Add database user` 创建一个用户。

> 注意：不要选中 `Make read-only`，因为我们有写数据库的操作。

最后分配给我们的类似下面的 mongodb url：

```
mongodb://<dbuser>:<dbpassword>@ds139327.mlab.com:39327/myblog
```

如我创建的用户名和密码都为 myblog 的用户，新建 config/production.js，添加如下代码：

config/production.js

```
module.exports = {
  mongodb: 'mongodb://myblog:myblog@ds139327.mlab.com:39327/mybl
og'
};
```

停止程序，然后以 production 配置启动程序:

```
NODE_ENV=production supervisor --harmony index
```

> 注意：Windows 用户安装 cross-env，使用：
>
> ```
> cross-env NODE_ENV=production supervisor --harmony index
> ```

4.15.2 pm2

当我们的博客要部署到线上服务器时，不能单纯的靠 `node index` 或者 `supervisor index` 来启动了，因为我们断掉 SSH 连接后服务就终止了，这时我们就需要像 pm2 或者 forever 这样的进程管理器了。pm2 是 Node.js 下的生产环境进程管理工具，就是我们常说的进程守护工具，可以用来在生产环境中进行自动重启、日志记录、错误预警等等。以 pm2 为例，全局安装 pm2：

```
npm install pm2 -g
```

修改 package.json，添加 start 的命令：

package.json

```
"scripts": {
  "test": "node --harmony ./node_modules/.bin/istanbul cover ./node_modules/.bin/_mocha",
  "start": "NODE_ENV=production pm2 start index.js --node-args='--harmony' --name 'myblog'"
}
```

然后运行 `npm start` 通过 pm2 启动程序，如下图所示 ：

```
> myblog@1.0.0 start /Users/nswbmw/Desktop/myblog
> NODE_ENV=production pm2 start index.js --node-args='--harmony' --name 'myblog'

[PM2] Starting /Users/nswbmw/Desktop/myblog/index.js in fork_mode (1 instance)
[PM2] Done.
```

App name	id	mode	pid	status	restart	uptime	cpu	mem	watching
myblog	0	fork	8075	online	0	0s	0%	11.8 MB	disabled

```
Use `pm2 show <id|name>` to get more details about an app
```

pm2 常用命令:

1. `pm2 start/stop` : 启动/停止程序
2. `pm2 reload/restart [id|name]` : 重启程序
3. `pm2 logs [id|name]` : 查看日志
4. `pm2 l/list` : 列出程序列表

更多命令请使用 `pm2 -h` 查看。

4.15.2 部署到 Heroku

Heroku 是一个支持多种编程语言的云服务平台，Heroku 也提供免费的基础套餐供开发者测试使用。现在，我们将论坛部署到 Heroku。

> 注意：新版 heroku 会有填写信用卡的步骤，如果没有请跳过本节。

首先，需要到 https://toolbelt.heroku.com/ 下载安装 Heroku 的命令行工具包 toolbelt。然后登录（如果没有账号，请注册）到 Heroku 的 Dashboard，点击右上角 New -> Create New App 创建一个应用。创建成功后运行：

```
$ heroku login
```

填写正确的 email 和 password 验证通过后，本地会产生一个 SSH public key，然后输入以下命令：

```
$ git init
$ heroku git:remote -a 你的应用名称
$ git add .
$ git commit -am "first blood"
$ git push heroku master
```

稍后，我们的论坛就部署成功了。访问：

```
https://你的应用名称.herokuapp.com/
```

4.15.3 部署到 UCloud

UCloud 是国内的一家云计算服务商，接下来我们尝试将博客搭在 UCloud 上。

> 小提示：不是给 UCloud 打广告。Heroku 不能用后，于是寻找可以免费试用的云主机，注册 UCloud 后发现没有免费试用，于是果断弃坑。过了一会 UCloud 的人打电话回访然后给充了点钱。。于是我就试了下。如果你们注册没有赠送金额，可以联系 UCloud 索要。。

创建主机

1. 注册 UCloud
2. 点击左侧的 `云主机`，然后点击 `创建主机`，统统选择最低配置
3. 右侧付费方式选择 `按时`（每小时），点击 `立即购买`
4. 在支付确认页面，点击 `确认支付`

购买成功后回到主机管理列表，如下所示：

业务组 全部 ＋ 创建主机

云主机控制台现已支持右键快捷菜单。您可以在主机列表项上单击右键，即可快捷地执行登录、重启、更改配置等操作。

主机名称	可用区	业务组	基础网络	配置	创建时间
nswbmw	北京二可用区B		内网IP:10.9.160.51 外网IP:106.75.47.229 BGP	1 1 0	2016-11-02 20:13:40

> 注意：下面所有的 ip 都替换为你自己的外网 ip。

环境搭建与部署

修改 config/production.js，将 port 修改为 80 端口：

config/production.js

```
module.exports = {
  port: 80,
  mongodb: 'mongodb://myblog:myblog@ds139327.mlab.com:39327/myblog'
};
```

登录主机，用刚才设置的密码：

```
ssh root@106.75.47.229
```

因为是 CentOS 系统，所以我选择使用 yum 安装，而不是下载源码编译安装：

```
yum install git #安装git
yum install nodejs #安装 Node.js
yum install npm #安装 npm

npm i npm -g #升级 npm
npm i pm2 -g #安装 pm2
npm i n -g #安装 n
n v6.9.1 #安装 v6.9.1 版本的 Node.js
n use 6.9.1 #使用 v6.9.1 版本的 Node.js
node -v
```

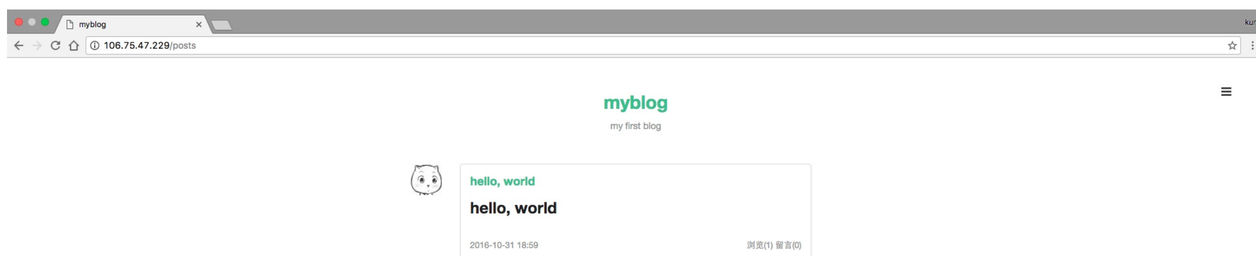
> 注意：如果 `node -v` 显示的不是 6.9.1，则断开 ssh，重新登录主机再试试。

此时应该在 `/root` 目录下，运行以下命令：

```
git clone https://github.com/nswbmw/N-blog.git myblog #或在本机 myblog 目录下运行 rsync -av --exclude="node_modules" ./ root@106.75.47.229:/root/myblog
cd myblog
npm i
npm start
pm2 logs
```

> 注意：如果不想用 git 的形式将代码拉到云主机上，可以用 rsync 将本地的代码同步到你的 UCloud 主机上，如上所示。

最后，访问你的公网 ip 地址试试吧，如下所示：



> 小提示：绑定域名不在本节讲解范围，读者可自行尝试。
>
> 小提示：因为我们选择的按时付费套餐，测试完成后，可在主机管理页面选择关闭主机，节约费用。

上一节：4.14 测试